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

Extended Version

Hanyu Zhao
Peking University
zhaohanyu@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 Centric Networking & Block Chain 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, for better performance 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*;

KEYWORDS

State machine replication, Geo-replication, Heterogeneity, Contention, Latency, Performance

*Corresponding author.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA
2018. ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267837

ACM Reference Format:
Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 16 pages.
https://doi.org/10.1145/3267809.3267837

1 INTRODUCTION

Today's applications commonly use replication to tolerate server failures and provide highly available services [11, 20]. The replication servers (*i.e.,* the *replica*s) coordinate via state machine replication protocols like Paxos [25] 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 [14, 19] has placed two critical demands on replication protocols: high throughput and low (wide-area) latency. Traditional single-leader protocols [26, 37, 38] 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 [33], 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 [29, 44]. Heterogeneity also exists in private data centers due to factors like multiple hardware generations, varying network conditions or competition for resources [8, 15, 21, 41].

Another approach is dynamically designating leaders [34], 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 [4, 5, 10, 40]. This problem can be even worse in the wide area: as requests take much longer time to finish, the probability of contention also rises [36].

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. In 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 is, 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 this extended version of the paper, we also provide a complete correctness proof of SDPaxos (Appendix A).

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 [26]) and multi-leader (Mencius [33], EPaxos [34]) 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.

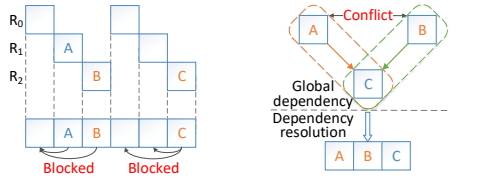
2 BACKGROUND AND MOTIVATION

State machine replication is a general approach to implementing fault-tolerant services [42]. 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 *learner*s 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 [26] 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. However, this single leader takes significantly heavier load than others, and inevitably becomes a performance bottleneck. Moreover,



(a) Static ordering (b) Dependency-based ordering

Figure 1: Two ordering approaches adopted by existing multi-leader protocols. Each square is a consensus instance, and the letter inside is the committed command. The arrows between squares in (b) represent dependencies, *e.g.*, $A \rightarrow C$ means C should be executed after A.

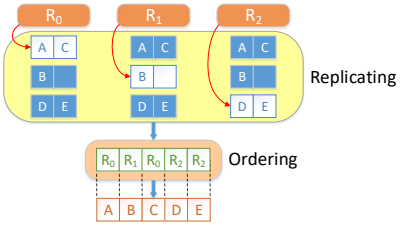


Figure 2: Separation of replicating and ordering.

in geo-replication, clients not co-located with the leader have to send requests to the remote leader, which brings significantly higher wide-area latency.

To address the single leader bottleneck, multi-leader protocols permit every replica to propose commands. However, Paxos's liveness property demands that there not be different replicas proposing in the same instance simultaneously. Therefore, each replica should commit commands in its own exclusive instance space, then we need to *order* the instances committed by different replicas.

A basic ordering approach is *static ordering*, which statically partitions the global instance space among replicas in a pre-established manner, *e.g.*, round-robin. This approach can hardly adapt to the difference and temporal variation of realtime speeds of replicas, resulting in high sensitivity to stragglers. Specifically, a replica can commit an instance only after it has committed all preceding instances, to ensure linearizability of concurrent requests [18]. Thus, if replicas open instances at different speeds, the front runners cannot commit their instances until the stragglers catch up. For example in Figure 1(a), there are *holes* in the global instance space in the absence of R_0's and R_1's commits, so R_1's and R_2's subsequent commits have to be blocked. As a result, all replicas have to run at the speed of the *slowest* one. Mencius [33], a representative static-ordering protocol, tackles this problem by replicas skipping their turns when they fall behind, however, it cannot fully eliminate this slow-down [34] (also confirmed in our evaluation).

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. This design minimizes the overhead of non-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 [34] 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 [7], 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_0, R_2, R_2 designated to the five global slots, we eventually get a totally ordered log of A, B, C, D, E.

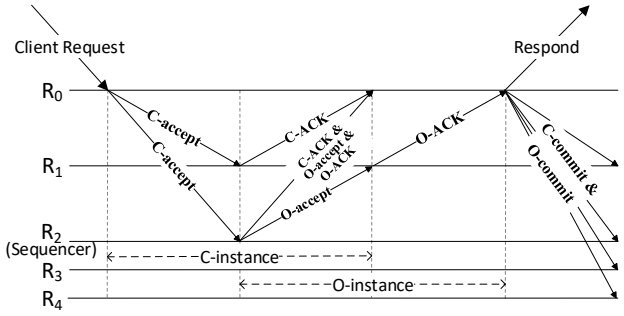


Figure 3: Message flow of the basic protocol of SDPaxos.

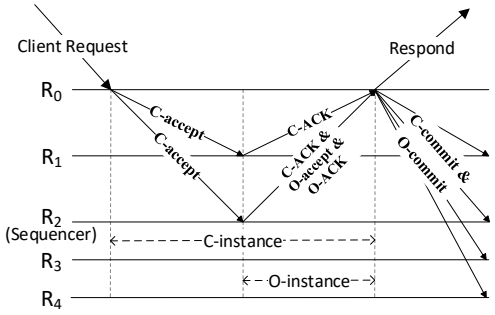


Figure 4: Message flow of the complete version of SDPaxos for five replicas. If the *C-accept* is not sent to the sequencer (R_2), there will be a *C-request* to it.

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 ordering 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 [5]. 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-accept*s to others, to assign this command to the jth global slot. Replicas will then accept these instances and send *C-ACK*s and *O-ACK*s 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-ACK*s and *O-ACK*s 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 [17].

Figure 5 shows the pseudocode of SDPaxos without failures. We assume the Prepare phases have been done beforehand. We denote the ith C-instance of R_n as C_{ni}, 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-accept*s (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.

1 **C-phase**
2 Replica R_n on receiving a client request for command α:
3 send *C-accept(n, i, α, bal_i)* to at least a majority
4 **if** *the C-accept is not sent to the sequencer* **then**
5 send *C-request(n, i)* to the sequencer
6 increment the C-instance counter i
7 Any replica R on accepting *C-accept(n, i, α, bal_i)*:
8 send *C-ACK(n, i, α, bal_i)* to R_n
9 R_n on receiving *C-ACK*s from a majority of replicas:
10 commit C_{ni} and broadcast *C-commit(n, i, α, bal_i)*
11 **O-phase**
12 Sequencer R_m on receiving *C-accept(n, i, α, bal_i)* or *C-request(n, i)* from R_n:
13 **if** $i \geq$ *number of O-instances for* R_n *in sequencer's assignment log* **then**
14 send *O-accept(j, n, bal_j)* to at least a majority including R_n
15 increment the O-instance counter j
16 Any replica R on accepting *O-accept(j, n, bal_j)*:
17 send *O-ACK(j, n, bal_j)* to R_n
18 **Ready condition for three or more than five replicas**
19 **if** R_n *receives O-ACKs from a majority of replicas* **then**
20 commit O_j and broadcast *O-commit(j, n, bal_j)*
21 **if** R_n *has committed* C_{ni} *and at least* i *O-instances for* R_n **then**
22 respond to client
23 **Ready condition for five replicas**
24 **if** R_n *is the sequencer* **then**
25 commit O_j and broadcast *O-commit(j, n, bal_j)* on receiving *O-ACK*s from a majority of replicas
26 **if** R_n *has committed* $C_{ni} \wedge$ *any* O_k *with* $k \leq j$ (O_j *is the* i*th O-instance for* R_n*) is accepted by a majority* **then**
27 respond to client
28 **else**
29 commit O_j and broadcast *O-commit(j, n, bal_j)* on receiving the *O-accept* from the sequencer
30 **if** R_n *has committed* $C_{ni} \wedge$ *any* O_k *with* $k \leq j$ (O_j *is the* i*th O-instance for* R_n*) is accepted by itself and the sequencer* **then**
31 respond to client

Figure 5: Pseudocode of SDPaxos in the absence of failures. Replicas accept a *C-accept* or *O-accept* if the ballot number (*bal*) is not lower than the highest number it has ever seen in the instance.

We say a command is *ready* [1] 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 *half a round trip* to achieve the final one-round-trip latency. The resulting message flow is shown in Figure 4.

[1] We do not use the term "commit a command" in the description of SDPaxos to avoid confusion with "commit an instance".

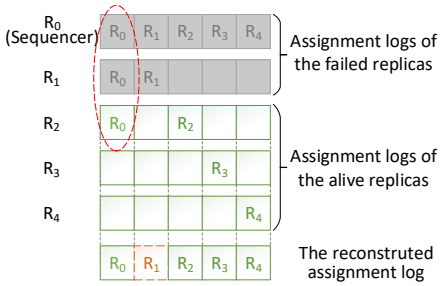


Figure 6: Recovery of O-instances for five replicas. R_0 (the old sequencer) and R_1 failed. O-instances for R_0 have been seen by a majority (R_0, R_1, R_2); those for R_2, R_3 and R_4 are seen by themselves. So we retry for R_1 in the hole, *i.e.*, the second O-instance. Note that this is just an intuitive schematic diagram; the real commit condition is stricter than this case.

Reducing latency under realistic conditions. In practice, replica groups tolerating one or two failures (*i.e.*, $N \leq 5$) are the most commonly used configurations [11, 14], 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 [34].)

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 ith 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-accept* 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 (R_2 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.,* R_0, R_2, R_3, R_4 in the figure; but there is still one replica left (R_1). 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 R_1 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-accept*s that the command leader has not seen on the latest *O-accept* sent to it. Similarly, *O-ACK*s that the sequencer has not seen are piggybacked on the *O-ACK* sent to it.

Finally, the command in proposed in C-instance C_{ni} will be executed in the jth slot by a replica after the C-instance and O-instance O_j are committed, and all preceding commands are executed—O_j is the ith O-instance for R_n 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, R_n, is suspected to have failed, another replica will initiate the recovery process. The replica will try to recover the C-instances of R_n, and also the O-instances if R_n was the sequencer. By default, we let R_{n+1} handle recovery for R_n (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 replies to *O-prepare*s, 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*s in the votes. Note that replicas only process messages for view change during this period. On receiving *vote*s 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 [18]. 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 be committed (otherwise at least alive replica will see it).

For five replicas, the "hole"s may be previously allocated to replica R_m and committed (R_m is the replica other than the old sequencer and the alive replicas who vote for the new sequencer). So we need to retry for R_m 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 R_m ($C[m]$ in Figure 7). $C[m]$ implies the maximum number of R_m's possibly ready commands, because the C-instance of any ready command of R_m must have been seen by one of the voters. Therefore, the sequencer should propose R_m's ID in the empty O-instances until there are at least $C[m]$ O-instances for R_m (lines 32 to 36). Although the procedure above implicitly assumes R_m also failed (because we need to tolerate two arbitrary failures), it works correctly even if R_m is still alive. A corner case is that, R_m commits an O-instance (*e.g.,* O_j) for it, but the corresponding C-instance (*e.g.,* C_{mi}) is not committed, then the new sequencer will propose *no-cl* if it does not see any value in both O_j and C_{mi}. This case won't violate consistency or linearizability, because the command in C_{mi} *must have not been executed or ready* (R_m will only be blocked at this slot). We allow R_m to accept the *no-cl* proposed by the new sequencer with a higher ballot number even if O_j 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".

```
Input: C[m]: number of C-instances of R_m accepted by the
       majority voters; O[m]: number of O-instances for R_m (R_m
       is the replica other than the old sequencer and the majority
       voters in five-replica groups)
if N == 5 then
    repeat
        propose R_m's ID in the first empty O-instance
        O[m] ← O[m] + 1
    until O[m] ≥ C[m] ∧ there is no empty O-instance preceding
      the C[m]^th one for R_m
foreach empty O-instance between non-empty ones do
    propose no-cl
```

Figure 7: Recovery of O-instances.

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 Appendix A.

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 there can be at most 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 chosen in each O-instance.

(ii) For five replicas, we focus the proof on 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_n. 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 α and β are commands on the same object, and the request for β arrives at the system after α is ready, then α will be executed before β.*

Proof sketch. Assume α is proposed in C-instance C_{ni} of R_n, while β is proposed in C-instance C_{mj} of R_m. When the sequencer proposes R_n's ID in the ith O-instance for R_n, there must be less than j O-instances for R_m in all preceding ones, because the request for the jth command of R_m (*i.e.,* β) has not arrived at the system yet. Now we prove that after α is ready, all O-instances preceding the ith one for R_n 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 after 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 $C[n] \leq O[n]$, there will be no hole preceding the ith O-instance for R_n.

By (i) and (ii), a sequencer can never assign β to a slot preceding the ith 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 $i_G \geq i_A \geq i_E$. Figure 8 illustrates the temporal relationship of i_G, i_A and i_E.

A replica sends *R-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-request*s. 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

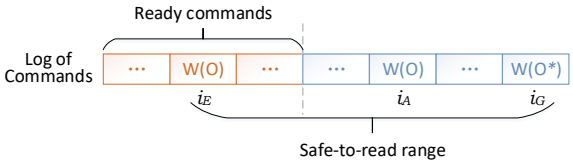


Figure 8: Temporal relationship of i_E, i_A and i_G. The number i_E points to the last update (write) on object O among all ready commands. A read on O can be safely performed after i_E. While i_A is the last sequence number allocated to an update on O (this command can have not be ready yet), and i_G is the last number allocated to any command (possibly on other objects, *e.g.*, O^* in the figure). Note that the three numbers are not necessarily inequal.

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 [12], to authorize the sequencer to directly reply to *R-request*s. 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.

4 IMPLEMENTATION AND OPTIMIZATIONS

We implemented SDPaxos on basis of an existing Go implementation of Multi-Paxos, Mencius and EPaxos, shared by the author of EPaxos [1]. We use these protocols to replicate an in-memory key-value store. Specifically, clients can issue read or write requests to the servers, then the server receiving a request will run the protocol to commit the command (for reads, SDPaxos uses the optimization rather than commit them). Then all servers will execute all the committed commands. The codes of all these protocols are built within a common framework, which minimizes the implementation-related performance difference. Below we describe our performance optimizations in this prototype implementation:

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-accept*s 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.

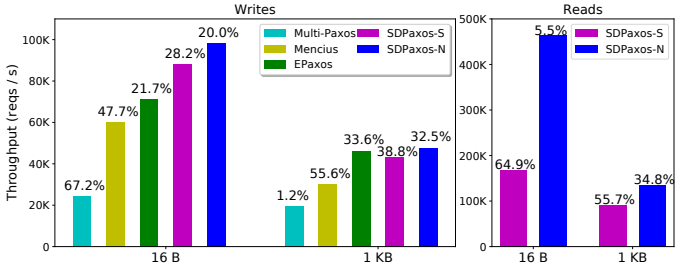
We implement a low-cost *on-demand* detection, in which replicas estimate the sequencer's healthiness according to whether the sequencer can satisfy their requirements for ordering. Specifically, each replica counts the number of *O-accept*s for itself received from the sequencer every 500 ms. If the 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.

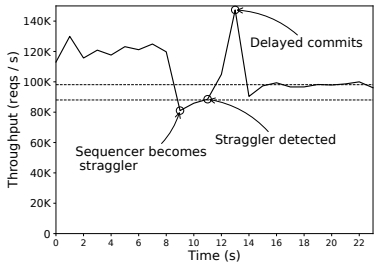
There are still many implementation-level optimization choices for SDPaxos. Below we explore several possible optimizations around the centralized ordering design:

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 [23], 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



(a) Throughput and reduction percentage without straggler detection.



(b) Throughput of SDPaxos after a slow sequencer is detected.

Figure 9: 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.

can fully eliminate the load imbalance of ordering. This machine and $N-1$ replicas can constitute a Paxos group for O-instances for one replica.

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.

5 EVALUATION

Experimental setup. We evaluate SDPaxos, Multi-Paxos [26], Mencius [33] and EPaxos [34], 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 straggler is the leader (otherwise the throughput is hardly affected). For SDPaxos, we let the sequencer and a non-sequencer replica be the straggler respectively to observe the difference. Figure 9(a) shows the throughput and the reduction percentage of these protocols with a straggler (the throughput without straggler is shown in §5.3, Figure 11).

With a non-sequencer straggler, SDPaxos always achieves the highest throughput (up to 1.6× and 1.4× 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 16 B 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

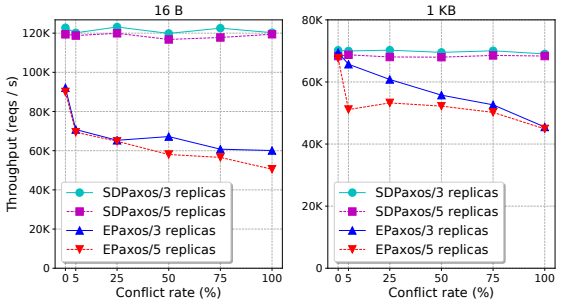


Figure 10: Throughput with increasing conflict rate.

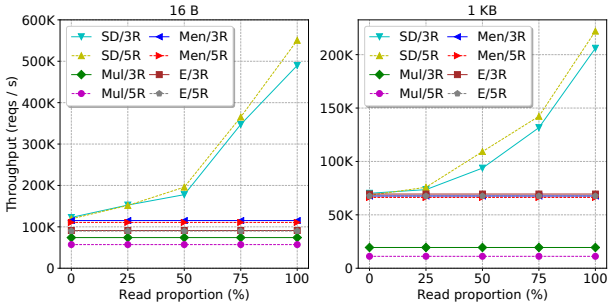


Figure 11: Throughput with no straggler and no conflict. "SD", "Mul", "Men", "E" represent SDPaxos, Multi-Paxos, Mencius and EPaxos respectively. "3R" and "5R" stand for three replicas and five replicas.

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.

	CA	OR	OH	IRE	SEL
CA	1.16	20	52	139	146
OR		0.02	68	125	133
OH			1.1	84	197
IRE				0.48	229
SEL					1.11

Table 1: The RTTs between the five regions (ms).

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

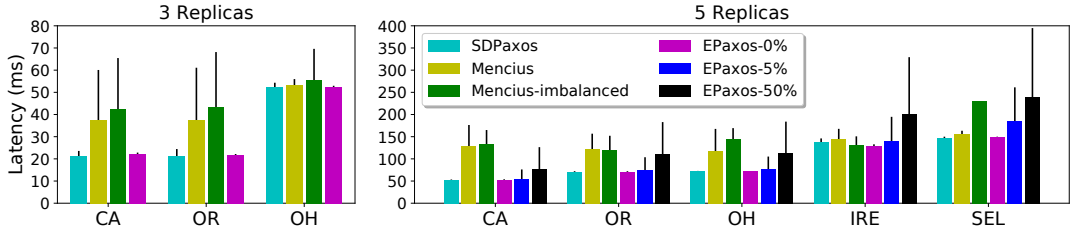


Figure 12: Average commit latency in each region. Error bars show 95% CI. "Imbalanced" means the number of clients in one region (OH and IRE for three and five replicas respectively) is half of that of the others. The percentage in EPaxos is conflict rate. The legend applies to both the left and the right subfigures (the same below).

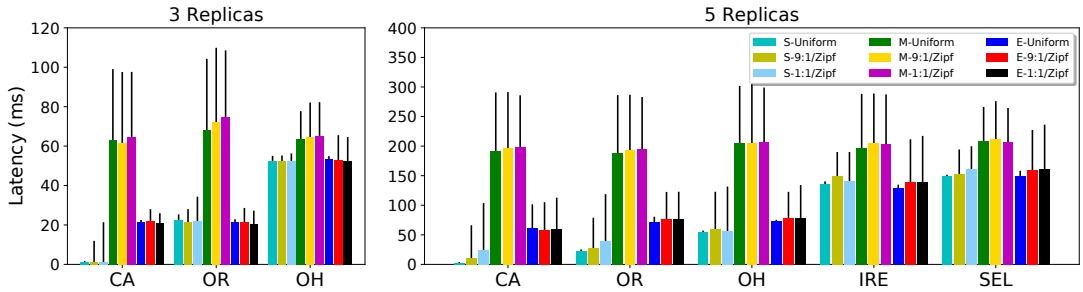


Figure 13: Average read latency in each region. Error bars show 95% CI. "S", "M", and "E" represent SDPaxos, Mencius and EPaxos respectively. "Uniform" and "Zipf" are key distributions. "9:1" and "1:1" are R/W ratios for Zipf (always 1:1 for Uniform).

the quorum. As is shown in Figure 12, SDPaxos always achieves optimal latency with three replicas. 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 [13] 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 500000 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 a 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 1:1 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.

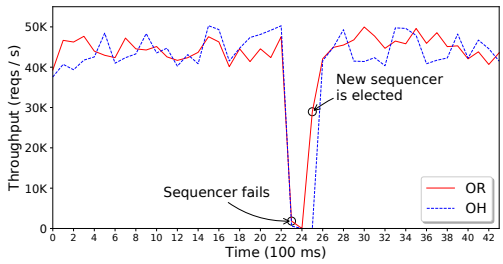


Figure 14: Throughput of the two non-sequencer replicas when the sequencer fails.

5.5 Service Availability under Failures

To evaluate the availability of SDPaxos, we measure the throughput of the non-sequencer replicas dynamically, during which the sequencer is shut down. We set this experiment in wide area (CA, OR and OH) since it will take higher latency to detect the sequencer's failure.

As shown in Figure 14, after the sequencer fails, throughputs of the other two replicas drop to zero for a short period. Then the one in OR is elected as the new sequencer. The theoretically highest latency of this period is the heartbeat interval and the sequencer lease duration plus the time of a view change. We set both the heartbeat interval and the lease duration to 500 ms, which guarantees the failover can always be finished within several hundreds of milliseconds. In Figure 14, the stalling period of the replica in OR and OH lasts about 300 ms and 400 ms respectively—the sequencer fails approximately at the middle of a heartbeat interval. The one in OH spends more time for the notification from the new sequencer.

6 RELATED WORK

Paxos, followed by a series of variants, has dominated the discussion of distributed consensus over years. Classic Paxos [25] commits a command in two rounds, and can be optimized to the one-round Multi-Paxos [26]. Fast Paxos [28] reduces latency with client requests sent directly to all acceptors, and employs a coordinator to handle conflicts, *i.e.,* requests received by replicas in different orders. Generalized Paxos [27] avoids conflicts by allowing non-interfering commands committed in different orders, and requires a leader to order interfering commands. Fast Paxos and Generalized Paxos need larger quorum size to reach consensus bypassing a proposer, resulting in higher wide-area latency. Viewstamped Replication [31, 37] and Raft [38] reach consensus at the level of the whole log rather than each instance, by employing a leader to replicate all log entries to other replicas.

Recent works focus on boosting performance of replication protocols. Mencius [33] and Egalitarian Paxos (EPaxos) [34] are two typical protocols designed for high throughput and low wide-area latency. SDPaxos draws on their multi-leader mechanism, and realizes efficient command ordering by the semi-decentralized design. In fact, similar communication pattern can be found in some works in related topics [7, 9, 16, 22]; however, they need either higher latency (at least two round trips [7, 22]), or stronger assumption (*e.g.,* reliable communication [9, 16]) to guarantee correctness. In particular, CORFU [7] is a distributed shared log which uses a centralized sequencer to assign log positions to clients dynamically. The key difference between CORFU and SDPaxos is that a CORFU client must first contact the sequencer, *then* write to the log, inducing higher latency. CORFU adopts this design because it uses a *static* mapping from logical positions in the log to physical pages—only after acquiring a log position can a client know which physical page to write. In contrast, in SDPaxos this mapping (*i.e.,* the assignment log, mapping log slots to commands) is generated dynamically, so replicas can replicate commands and ask for ordering in parallel.

Speculative Paxos [39] and NOPaxos [30] further show that command ordering can be done with very low overhead in the network layer, hence nearly zero coordination needed for the replication protocol layer. However, such designs rely on software-defined networking or next-generation hardware in data centers.

Some Multi-Paxos systems use master lease [12] to facilitate the leader to perform local reads. Megastore [6] allows every replica to read all objects locally, at the expense of poor write performance. Paxos quorum leases [35] facilitate local reads on a subset of objects, which needs an appropriate allocation of leases on objects to replicas. SDPaxos is a natural fit for quorum lease as the sequencer can easily know which replica read an object the most often.

Replication protocols are the cornerstone of a broad range of fault-tolerant services. For example, Boxwood [32], Chubby [11] and Apache ZooKeeper [20] use replication protocols to provide available distributed coordination services. Google's Spanner [14] and Megastore [6] are distributed storage systems using Paxos to replicate data. Calvin [43] uses ZooKeeper to commit transactions in deterministic distributed databases. MDCC [24], TAPIR [45] and Janus [36] merge replication protocol and concurrency control protocol in geo-distributed transaction processing.

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 [3].

A PROOF OF CORRECTNESS

A.1 Problem Definition

SDPaxos guarantees the following safety properties, which are similar to those provided by the other Paxos-based protocols:

Nontriviality Any command executed in any slot of the global log must have been proposed by a client.

Consistency At most one command can be executed by different replicas in the same slot of the global log.

Linearizability If two commands α and β are operations on the same object, and β is proposed by a client after the request for α is responded to by any replica (both are eventually responded to), then α will be executed before β by every replica.

Now we prove these properties for SDPaxos.

A.2 Proof

First, we prove that C-instances satisfy the following properties.

Lemma A.1. *Any command committed in any C-instance must have been proposed by a client, and at most one command can be committed in the same C-instance.*

Proof. C-instances run classic Paxos algorithm. Paxos satisfies non-triviality and consistency, which has been proved.

Theorem A.2 (Nontriviality). *Any command executed in any position of the global log must have been proposed by a client.*

Proof. Any command in any position of the global log must have been committed in some C-instance. By Lemma 1, C-instances only commit proposed commands.

For consistency and linearizability, there are two cases: one is 3 or more than 5 replicas, the other is 5 replicas. We will discuss the two case respectively. Note that in this paper, we say a command is "committed" when it is "ready" in the SDPaxos paper, for simplicity.

Lemma A.3. *In SDPaxos for 3 or more than 5 replicas, at most one replica can be committed in an O-instance.*

Proof. In SDPaxos for 5 replicas, O-instances also run classic Paxos algorithm. Similarly to the proof of Lemma 1.1, Paxos ensures consistency.

Theorem A.4 (Consistency for 3 or more than 5 replicas). *At most one command can be executed by different replicas in the same position of the global log.*

Proof.

⟨1⟩1. Suffices Assume: i is a positive integer and any O-instance O_j with $j \leq i$ has been committed. The value chosen in O_i is n.
Prove: There can be at most one command in the i^{th} slot of the global log.

Proof: Since a command in the ith slot of the global log can be executed only after every O-instance O_j with $j \leq i$ is committed, it suffices to show that there can be at most one command in the ith slot of the global log after all preceding O-instances have been committed.

⟨1⟩2. The number of times that n is chosen in all O-instances O_j with $j \leq i$ is a determined value.

Proof: By Lemma 1.2, there will be a determined value chosen in each O-instance after committed. Therefore, the number of times that n is chosen is also determined.

⟨1⟩3. Assume: O_i is the k^{th} O-instance in which n is chosen.
Prove: At most one command can be chosen in C-instance C_{nk}.

Proof: By Lemma 1.1.

⟨1⟩4. Q.E.D.

Proof: By ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3.

Theorem A.5 (Linearizability for 3 or more than 5 replicas). *If α and β are commands on the same object, and β is requested by client after α is committed (both are eventually committed), then α will be executed before β.*

Proof.

⟨1⟩1. Assume: α is proposed in C-instance C_{ni}.
Prove: At the moment when α is committed, there must be at least i O-instances for R_n in the sequencer's assignment log.

Proof: We prove this contradiction. Assume there are fewer than i O-instances for R_n in the sequencer's assignment log when α is committed. The only possible case is, the sequencer failed and is reelected, and there are fewer than i O-instances for R_n seen by a majority by that time. However, since the command in C_{ni} has been committed, there must be i O-instances for R_n seen by a majority when the command is committed. Then the O-instances for R_n that the sequencer did not see in the view change must be proposed later by the new sequencer (because they will be rejected with a stale view number). Because there is a contradiction, the assumption that there are fewer than i O-instances for R_n in the sequencer's assignment log must be false.

⟨1⟩2. At the moment when the request for β is received, there must be at least i O-instances for R_n in the sequencer's assignment log.

⟨2⟩1. Case: The sequencer has never failed after α is committed.
Proof: By ⟨1⟩1, there must be at least i O-instances for R_n in the sequencer's assignment log. Because the O-instance counter increases monotonically, there must be at least i O-instances for R_n in the sequencer's assignment log when β is received.

⟨2⟩2. Case: The sequencer has ever failed and been reelected after α is committed.
Proof: Because α has been committed, there must be at least i O-instances for R_n that have been accepted by a majority. Whenever a sequencer is elected in a view change, it can see at least i O-instances for R_n in its assignment log.

⟨2⟩3. Q.E.D.
Proof: Cases ⟨2⟩1 and ⟨2⟩2 are exhaustive.

⟨1⟩3. Q.E.D.
By ⟨1⟩1, ⟨1⟩2, and the fact that a sequencer will never assign a command to a slot preceding a non-empty one in its assignment log.

Now we discuss the case of 5 replicas, where O-instances are not committed by Paxos. We say an O-instance is committed in the k^{th} view, if the learner commits this instance on receiving an *O-ACK* with a view number of k. For simplicity, we assume there is a sequencer elected with every view number, because there will be no O-instance committed in a view without a elected sequencer.

Theorem A.6 (Consistency for 5 replicas). *At most one command can be executed by different replicas in the same position of the global log.*

Proof.

⟨1⟩1. Suffices Assume: A replica R executes a command α proposed in C-instance C_{ni} in the j^{th} global slot. It commits O-instance O_j in the v^{th} view.
Prove: Every replica can only commit n in O_j. And for every replica, after all preceding O-instances are committed, O_j will be the i^{th} O-instance for R_n.

Proof: Since C-instance executes classic Paxos algorithm, every replica can only commit α in C_{ni}. If a replica commits α in C_{ni}, and commit n in O_j as the i^{th} O-instance for R_n, it will also execute α in the j^{th} global slot.

⟨1⟩2. If the sequencer of the v^{th} view never fails, then consistency always holds.

Proof: For those O-instances that are preceding O_j and committed by R in the v^{th} view, other replicas will commit the same result if the sequencer of the v^{th} view never fails. For those O-instances that are preceding O_j and committed by R in previous views, they have been accepted by a majority in the $(v-1)^{th}$ view change, so other replicas will also commit the same result. Because all O-instances preceding O_j must be committed by R in the v^{th} or some previous view, other replicas will commit the same result in all those O-instances.

⟨1⟩3. Whenever a sequencer is elected in the $(v')^{th}$ view change ($v' > v$), the values in all O-instances O_k with $k \leq j$ in the new sequencer's assignment log must be the same as those in R's.

⟨2⟩1. Case: The replica R is one of the majority voters.

Proof: The replica R and other voters will report all these O-instances in its vote. All these O-instances in these votes must have the same values: for those O-instances that are preceding O_j and committed by R in the v^{th} view, they are proposed by the same proposer; for those O-instances that are preceding O_j and committed by R in previous views, they have been accepted by a majority in the $(v-1)^{th}$ view change. Therefore, the new sequencer must see the same values in these O-instances as R.

⟨2⟩2. Case: The replica R is not one of the majority voters.

⟨3⟩1. The values in all O-instances O_k with $k \leq j$ for the majority voters in the new sequencer's assignment log must be the same as those in R's.

Proof: Similarly to ⟨2⟩1, for any one of the majority voters, the new sequencer will see all O-instances for it.

⟨3⟩2. The values in all O-instances O_k with $k \leq j$ for the sequencer of the v^{th} view in the new sequencer's assignment log must be the same as those in R's.

Proof: Those O-instances committed in the v^{th} view must have been accepted by a majority. While those O-instances committed in previous views must also have been accepted by a majority in the $(v-1)^{th}$ view change. So the new sequencer will see the same values in all O-instances for the sequencer of the v^{th} view.

⟨3⟩3. Q.E.D.

Proof: By ⟨3⟩1 and ⟨3⟩2, all O-instances O_k with $k \leq j$ for all replicas except one replica (say R_m) can be traced by the new sequencer. So the gaps in the new sequencer's assignment must be originally allocated to R_m. Because R has executed the command in C_{ni} and all preceding ones, which the corresponding C-instances must have been accepted by a majority, the new sequencer will make sure that there are at least the same number of O-instances for R_m as that in R's assignment log. Therefore, the values in all O-instances O_k with $k \leq j$ in the new sequencer's assignment log must be the same as those in R's.

⟨2⟩3. Q.E.D.

Proof: Cases ⟨2⟩1 and ⟨2⟩2 are exhaustive.

⟨1⟩4. Q.E.D.

Proof: By ⟨1⟩1, ⟨1⟩2 and ⟨1⟩3.

Theorem A.7 (Linearizability for 5 replicas). *If α and β are commands on the same object, and β is requested by client after α is committed (both are eventually committed), then α will be executed before β.*

Proof.

⟨1⟩1. Assume: α is proposed in C-instance C_{ni}.
Prove: At the moment when α is committed, there must be at least i O-instances for R_n without preceding hole (empty ones between non-empty ones) in the sequencer's assignment log.

Proof: We prove this by contradiction. Assume there are fewer than i O-instances for R_n or with preceding hole in the sequencer's assignment log when α is committed. The only possible case is, the sequencer failed is reelected, R_n is not one of the majority voters, and C_{ni} has not been seen by a majority by that time. (If R_n is a voter, then the new sequencer will see all its O-instances; otherwise, the new sequencer will ensure there will be at least i O-instances for R_n with no preceding hole, according to the view change protocol.) However, since the command in C_{ni} has been committed, C_{ni} must has been seen by a majority when the command is committed. After the new sequencer is elected, C_{ni} will be rejected due to a stale view number. Then R_n will contact the new sequencer and synchronize all the O-instances, so it will change the value in its i^{th} O-instances for itself to *no-cl*. Here we get a contradiction, hence the assumption that there are fewer than i O-instances for R_n or with preceding gaps in the sequencer's assignment log must be false.

⟨1⟩2. At the moment when the request for β is received, there must be at least i O-instances for R_n without preceding hole in the sequencer's assignment log.

⟨2⟩1. Case: The sequencer has never failed after α is committed.

Proof: By ⟨1⟩1, there must be at least i O-instances for R_n in the sequencer's assignment log. Because the O-instance counter increases monotonically, there must be at least i O-instances for R_n without preceding hole in the sequencer's assignment log when β is received.

⟨2⟩2. Case: The sequencer has ever failed and been reelected after α is committed.

⟨3⟩1. Case: R_n is one of the majority voters.

Proof: Since R_n has committed the command α, it must have seen at least i O-instances for itself with no preceding hole. Then the new sequencer will see all these O-instances in R_n's vote.

⟨3⟩2. Case: R_n is the old sequencer.
Proof: Since R_n has committed the command α, then all preceding O-instances must have been seen by a majority. Then the new sequencer will see all these O-instances in the view change.
⟨3⟩3. Case: R_n is neither a voter nor the old sequencer.
Proof: Since R_n has committed the command α, C_{ni} must have been seen by a majority. So the new sequencer will make sure there are at least i O-instances for R_n with no preceding hole, *i.e.,* by proposing n in the gaps.
⟨3⟩4. Q.E.D.
Proof: Cases ⟨3⟩1, ⟨3⟩2 and ⟨3⟩3 are exhaustive.
⟨2⟩3. Q.E.D.
Proof: Cases ⟨2⟩1 and ⟨2⟩2 are exhaustive.
⟨1⟩3. Assume: β is proposed in C-instance C_{mj}. The sequencer has ever failed and been reelected after it proposes m in an O-instance as the j^{th} one for R_m.
Prove: The new sequencer will not propose m in an O-instance preceding the i^{th} one for R_n.
⟨2⟩1. Case: R_n is one of the majority voters.
Proof: Since R_n has committed the command α, it must have seen at least i O-instances for itself with no preceding hole. Then the new sequencer will see all these O-instances in R_n's vote. So it cannot propose m in an O-instance preceding the i^{th} one for R_n.
⟨2⟩2. Case: R_n is the old sequencer.
Proof: Since R_n has committed the command α, then all preceding O-instances must have been seen by a majority. Then the new sequencer will see all these O-instances in the view change. So it cannot propose m in an O-instance preceding the i^{th} one for R_n.
⟨2⟩3. Case: R_n is neither a voter nor the old sequencer.
Proof: Since R_n has committed the command α, C_{ni} must have been seen by a majority. So the new sequencer will make sure there are at least i O-instances for R_n with no preceding hole, *i.e.,* by proposing n in the gaps. So it cannot propose m in an O-instance preceding the i^{th} one for R_n.
⟨2⟩4. Q.E.D.
Proof: Cases ⟨2⟩1, ⟨2⟩2 and ⟨2⟩3 are exhaustive.
⟨1⟩4. Q.E.D.
By ⟨1⟩1, ⟨1⟩2, and ⟨1⟩3.

Finally, our system also satisfies the following property:

Liveness Any proposed command will eventually be executed by every non-faulty replica, given that at least a majority of replicas are non-faulty, clients and replicas keep resending messages and messages are eventually delivered to their destinations, and a new sequencer will eventually be elected after the old one's failure.

The assumption that sequencer elections always make progress can be realized by setting randomized election timeouts for different replicas, in which non-sequencer replicas will detect sequencer's failure sporadically. This design prevents an election from ending up with no winner, since different candidates may compete for votes.

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.

REFERENCES

[1] 2013. EPaxos code base. (2013). https://github.com/efficient/epaxos
[2] 2018. Amazon Elastic Compute Cloud. (2018). https://aws.amazon.com/ec2/
[3] 2018. Source code of SDPaxos. (2018). https://github.com/zhypku/SDPaxos
[4] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-contention Workloads. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 121–134. https://doi.org/10.14778/3149193.3149194
[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766
[6] 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 *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf
[7] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=2228298.2228300
[8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. ACM, New York, NY, USA, 242–253. https://doi.org/10.1145/2018436.2018465
[9] Kenneth Birman, André Schiper, and Pat Stephenson. 1991. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.* 9, 3 (Aug. 1991), 272–314. https://doi.org/10.1145/128738.128742
[10] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, Vol. 1. 126–134 vol.1. https://doi.org/10.1109/INFCOM.1999.749260
[11] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 335–350. http://dl.acm.org/citation.cfm?id=1298455.1298487
[12] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*. ACM, New York, NY, USA, 398–407. https://doi.org/10.1145/1281100.1281103
[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152
[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-distributed Database. (2012), 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905
[15] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[16] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.* 36, 4 (Dec. 2004), 372–421. https://doi.org/10.1145/1041680.1041682

[17] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[18] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[19] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An Analysis of Facebook Photo Caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 167–181. https://doi.org/10.1145/2517349.2522722

[20] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[21] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware Distributed Parameter Servers. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 463–478. https://doi.org/10.1145/3035918.3035933

[22] M. F. Kaashoek and A. S. Tanenbaum. 1991. Group communication in the Amoeba distributed operating system. (May 1991), 222–230. https://doi.org/10.1109/ICDCS.1991.148669

[23] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[24] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[25] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

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

[27] Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report. 60 pages. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

[28] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (October 2006), 79–103. https://www.microsoft.com/en-us/research/publication/fast-paxos/

[29] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/1879141.1879143

[30] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 467–483. http://dl.acm.org/citation.cfm?id=3026877.3026914

[31] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report. Technical Report MIT-CSAIL-TR-2012-021, MIT.

[32] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. 2004. Boxwood: Abstractions As the Foundation for Storage Infrastructure. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=1251254.1251262

[33] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 369–384. http://dl.acm.org/citation.cfm?id=1855741.1855767

[34] 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 (SOSP '13)*. ACM, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[35] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2014. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 22, 13 pages. https://doi.org/10.1145/2670979.2671001

[36] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits Under Conflicts. (2016), 517–532. http://dl.acm.org/citation.cfm?id=3026877.3026917

[37] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC '88)*. ACM, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.62549

[38] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[39] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 43–57. http://dl.acm.org/citation.cfm?id=2789770.2789774

[40] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 401–417. http://dl.acm.org/citation.cfm?id=3026877.3026909

[41] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 7, 13 pages. https://doi.org/10.1145/2391229.2391236

[42] Fred B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[43] 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 (SIGMOD '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[44] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 29–42. http://dl.acm.org/citation.cfm?id=1855741.1855744

[45] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404