DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications

Faisal Nawab
University of California, Santa Cruz
Santa Cruz, CA
fnawab@ucsc.edu

Divyakant Agrawal
Amr El Abbadi
University of California, Santa Barbara
Santa Barbara, CA
[agrawal,amr]@cs.ucsb.edu

ABSTRACT

In this paper, we propose *Dynamic Paxos* (DPaxos), a Paxos-based consensus protocol to manage access to partitioned data across globally-distributed datacenters and edge nodes. DPaxos is intended to implement a State Machine Replication component in data management systems for the edge. DPaxos targets the unique opportunities of utilizing edge computing resources to support emerging applications with stringent mobility and real-time requirements such as Augmented and Virtual Reality and vehicular applications. The main objective of DPaxos is to reduce the latency of serving user requests, recovering from failures, and reacting to mobility. DPaxos achieves these objectives by a few proposed changes to the traditional Paxos protocol. Most notably, DPaxos proposes a *dynamic* allocation of quorums (*i.e.*, groups of nodes) that are needed for Paxos Leader Election. Leader Election quorums in DPaxos are smaller than traditional Paxos and *expand* only in the presence of conflicts.

ACM Reference Format:
Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications . In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196928

1 INTRODUCTION

The utilization of edge nodes is inevitable for the success and growth of emerging low latency applications, such as Augmented and Virtual Reality (AR/VR) and vehicular networks. Such applications have stringent latency requirements that the current cloud model cannot satisfy. This is due to the large communication latency between users and their closest datacenter (up to 100ms [44]). This latency problem is exacerbated for applications that serve users across large geographical areas. In such cases, users incur wide-area latency as large as 100s of milliseconds to seconds. Placing data closer to users at edge nodes overcomes this fundamental communication latency limit. We envision, as others [7], that the cloud model will extend to edge locations similar to how content delivery networks utilize edge locations. However, rather than edge locations being used for data caching only, they will also host data management components that will allow manipulation and querying of local edge partitions. In this paper, we focus on transaction processing as the data management task to be supported by the edge data management components. The model and focus of this paper aims to serve web and cloud applications, such as online shops, social networks, and collaborative applications.

In this paper, we propose *Dynamic Paxos* (DPaxos), a Paxos-based consensus protocol [21, 22]. DPaxos is intended to be used as the State Machine Replication (SMR) component in data management systems for the edge. In such systems, we envision utilizing edge nodes around the world by partitioning the data and placing each data partition at the edge node closest to its corresponding users. DPaxos, as a SMR component, manages access to data partitions in the form of consistent and fault-tolerant transactions and commands. DPaxos targets harnessing the benefits of deploying on edge nodes by supporting: (1) access locality: requests are served from a nearby edge node, (2) data mobility: partition copies follow moving users in real-time, and (3) flexible fault-tolerance: nearby edge nodes can be used to recover from failures.

We build upon Paxos due to its wide adoption in both academia and industry and its use on various data management applications, such as transaction management [5, 18, 31, 40, 41], atomic broadcast [8, 17], consistent replication [10, 13, 45, 46], and stream processing [3]. Therefore, an improved and specialized Paxos protocol (*i.e.*, DPaxos) impacts a wide-range of data management applications.

Traditional Paxos protocols [9, 21, 22] and Paxos variants for geo-replication [6, 18, 32, 33, 41, 48] do not explicitly consider resources beyond datacenters. This makes the large wide-area latency inevitable. Straightforward solutions to deploy existing Paxos variants on edge resources are also inefficient. Paxos variants rely on majority-based techniques (*i.e.*, coordination is performed by communicating with a majority of nodes). In a system with a large number of nodes, such as the edge, majority-based approaches are prohibitive, since they entail communication with a majority of a possibly massive number of nodes for each step. We discuss the shortcomings of other existing approaches in more details in Section B.1.

DPaxos proposes **Zone-centric Quorums** as an alternative to majority-based techniques to avoid unnecessary wide-area communication. A zone denotes a collection of neighboring edge nodes. DPaxos restricts the communication corresponding to a data partition to be within the zones where its users are located. To do this, DPaxos distinguishes between the quorums that are needed to

SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196928

perform the main two tasks in Paxos: *Leader Election* (coordination with other nodes to select a leader for a partition and is typically invoked in reaction to failures or mobility) and *Replication* (committing data from a leader to secondary nodes and is typically invoked for every transaction or request). In typical workloads, Replication is more frequent than Leader Election, and thus we prioritize optimizing its performance. Ideally, for performance, Replication would be performed within a zone rather than a majority of nodes. Flexible Paxos—proposed by Howard et. al. [16] and adapted for wide-area replication in WPaxos [2]—shows that it is possible to assign arbitrarily small Replication quorums as long as they satisfy the condition: a Leader Election quorum must intersect *all* Replication quorums. This means that in Flexible Paxos-based approaches, *the trade-off of small Replication quorums within zones is an expensive Leader Election quorum that must span all zones.*

We base DPaxos' Zone-Centric Quorums on the theoretical foundation laid by Flexible Paxos and adapt its quorum allocation techniques to the practical application of data management on globally-distributed edge nodes. Then, we propose two approaches to overcome Flexible Paxos' significant Leader Election penalty: **(1) Expanding Quorums:** this approach overcomes Flexible Paxos' intersection condition and allows both Leader Election and Replication quorums to be small. DPaxos is the first Paxos protocol that allows Leader Election to *not* intersect with all Replication quorums. Rather, the Leader Election quorum starts small and then grows to only intersect with Replication quorums that are being used by other leaders. **(2) Leader Handoff:** this approach supports fast leader mobility. Mobility, unlike failures, is triggered by known user actions, and hence can be exploited to optimize Leader Election. DPaxos exploits this to enable Leader Election via a lightweight, single round of messaging between the old and the new leaders.

The main contribution of DPaxos is that it introduces a design space of Paxos protocols where Leader Election quorums are dynamic (*i.e.*, expanding) rather than statically defined. We take this concept of Expanding Quorums and study the challenges and opportunities in realizing it by designing DPaxos and experimenting with various methods of Expanding Quorums. In particular, we find that combining Expanding Quorums with Flexible Paxos quorums [16] is a powerful idea. Also, we find that Expanding Quorums can take many shapes with different trade-offs. However, we also face challenges due to the increased complexity of DPaxos compared to other Paxos variants. This complexity leads to the need of maintaining more state information at nodes that can accumulate and cause an overall degradation of performance if left unhandled. In this paper, we show how we face these challenges while maintaining the performance improvements of DPaxos. Finally, we provide discussions on the safety of DPaxos and handling the garbage collection of accumulating state.

We present relevant background about Paxos in Section 2 and the system model in Section 3. The design and implementation of DPaxos is proposed in Section 4. Section 5 shows the performance evaluation. The paper concludes in Section 6 with a summary. Additional experimental evaluations and related work are presented in Sections A and B.

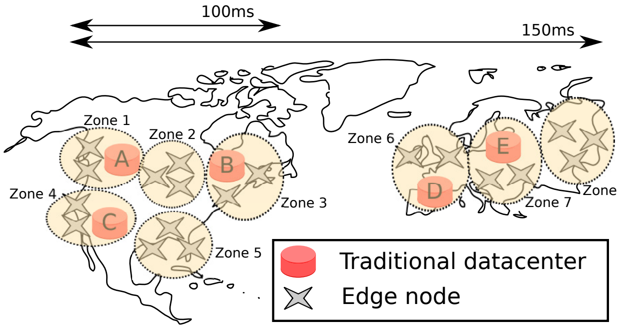


Figure 1: An example of an edge environment and DPaxos zones. A zone represents a disjoint set of neighboring nodes.

2 PAXOS BACKGROUND

Paxos [9, 21, 22] is a consensus algorithm to solve the problem of *deciding* (also called choosing) a single value among proposals by multiple nodes in a fault-tolerant manner. In many data management systems, Paxos is used to coordinate access to data by making it *decide* which transaction or request is committed among conflicting contenders.

Paxos resolves concurrent requests (also called proposals) using unique *proposal ids*. In the Leader Election phase, a proposal is assigned a unique id, p, and a prepare(p) message is sent to a majority. Assume that node A sent the prepare message. Processes receiving the prepare reply with a promise() message if p is the highest proposal id among the ones they have already received. With the promise, the most recent accepted proposal q, if any, and its value v_q are also sent. Process A is considered to be elected a leader if it receives a majority of promises for p. Then, A proceeds to the Replication phase with the proposal p', which corresponds to the highest received proposal, q. Otherwise, A proceeds with its own proposal if no proposals were sent along with promise() messages. In the Replication phase, the value v associated with p' is sent in a propose(p',v) message. A node replies with an accept(p') message if the proposal id is greater than or equal to the highest promised proposal. Once A receives a majority of accept(p') messages, then the value v is decided. If any step fails throughout this algorithm, a node might retry again with a higher proposal number.

In practice, Paxos is used to decide a sequence of values in a log, where each position is called a *slot* or *entry*. Multi-Paxos [22] is an efficient variant of Paxos that optimizes for this case. Rather than performing the Leader Election phase for every slot independently, a leader is elected for a subset of slots that potentially cover all remaining slots. In such a case, we call the leader a *prolonged leader*. Requests would be directed to the prolonged leader, and the prolonged leader *bypasses the Leader Election phase*, thus reducing the number of needed phases from two to one. However, in the case of a leader failure or needing to change the location of the leader, a new Leader Election round is necessary to elect the new leader. Thus, even while using Multi-Paxos, mobile applications may regularly incur the overhead of Leader Election. Since Multi-Paxos is the most prevalent in practice, we focus on it when developing DPaxos.

3 SYSTEM MODEL

In this section, we present the architecture and system model we consider.

A globally-distributed edge model. DPaxos utilizes globally-distributed datacenters and edge nodes around the world (depicted in Figure 1). We will use the terms *node*, *replica* and *datacenter* interchangeably to denote both traditional and edge datacenters and the term *zone* to denote a disjoint set of nodes (zones are defined by the system administrator). Traditional datacenters are large-scale datacenters that have the capacity to host tens of thousands of machines. Edge nodes, on the other hand, denote emerging edge-datacenter technologies, such as cloudlets and micro datacenters that are smaller than traditional datacenters.

Mobility. The location where users access a partition may change frequently and rapidly, especially in mobile applications such as vehicular applications. A data partition's copies corresponding to mobile users must follow users and migrate to the edge node that is closest to their new location. In most cases, a user moving from one location to another will only incur a small difference in latency to communicate to the edge node hosting his or her partition. This makes the action of moving data from the old location to the new location a non-imminent issue. However, when the partition eventually migrates, doing so more efficiently (with lower latency and communication overhead) would have a less drastic impact on the overall performance.

Fault-tolerance model. We distinguish between two types of failures: individual datacenter outages and natural disasters (zone-scale failures). The frequency of individual datacenter outages is much higher than zone-scale failures. A recent study showed that outages that affect more than one datacenter (*i.e.*, one or more zones) at once amounts for only 3% of all datacenter outages [15]. The difference in the type and frequency of failures invites a nuanced definition of fault-tolerance. Rather than the traditional definition of fault-tolerance as the number of tolerated failures, f, we specify two values: the number of tolerated individual datacenter failures in each zone, f_d, and the number of tolerated zone failures, f_z. In the rest of this paper, we assume that the number of edge datacenters in each zone is at least $2f_d + 1$ and that the number of zones is at least $2f_z + 1$. DPaxos adopts a flexible fault-tolerance model, where the level of fault-tolerance (f_d and f_z) can be configured by the user.

4 DPAXOS DESIGN

In this section, we propose DPaxos. We begin with an overview of DPaxos in Section 4.1 followed by a detailed description of DPaxos proposals.

4.1 DPaxos Overview

DPaxos' goal is to optimize latency of deciding values to slots in a globally-distributed edge system. This is achieved by reducing the size of Leader Election and Replication quorums which leads to less wide-area communication. Reducing the size of quorums also results in low communication overhead. DPaxos introduces three techniques to reduce quorum sizes:

- (1) *Zone-centric Quorums:* this method redefines Leader Election and Replication quorums to make the Replication phase free from unnecessary, inter-zone communication and enable flexible control of fault-tolerance guarantees. In DPaxos, Replication quorums are defined to be as small as possible (only consisting of $f_d + 1$ nodes in $f_z + 1$ zones even if the number of all nodes, n, is much larger than the size of the Replication quorum, $\mathcal{F} = (f_d+1)\times(f_z+1)$). These Replication quorums are much smaller than majority quorums in the environments we consider where n is much larger than the $\mathcal{F}$. To accommodate for such small Replication quorums, larger Leader Election quorums are needed. In general, the restriction for quorum allocations is that any Leader Election quorum must intersect all Replication quorums as shown in a recent work by Howard et. al. [16]. We will call this restriction the **inter-intersection** condition to denote the requirement for a Leader Election quorum to intersect with all Replication quorums. When Replication quorums are set to be small (our goal), this makes Leader Election quorums large. Overcoming the inter-intersection restriction [16] is the topic of the next two techniques.
- (2) *Expanding Quorums:* this method overcomes the restricting inter-intersection condition, which states that Leader Election quorums must intersect with all Replication quorums). Rather, Expanding Quorums requires that Leader Election quorums must only intersect with themselves and hence they do not need to intersect with any Replication quorums. We call this the **intra-intersection** condition as opposed to the inter-intersection condition. Expanding Quorums stems from the observation that a Leader Election quorum only needs to intersect with *concurrent* Replication quorums—not all possible Replication quorums. DPaxos leverages this observation by making aspiring leaders *announce* the Replication quorums they intend to use. Concurrent aspiring leaders will detect ongoing Replication quorums through these announcements. If such Replication quorums are detected, then the aspiring leader expands its Leader Election quorum to intersect with them. DPaxos proposes two quorum types that implement Expanding Quorums: (2.a) Delegate Quorums, where the size of a Leader Election quorum is only a majority of zones rather than all zones (even for single-zone Replication quorums). (2.b) Leader Zone Quorums, where the size of a Leader Election quorum is a single zone only in the typical case.
- (3) *Leader Handoff:* with this methods, a leader can relinquish leadership to another node with a single light-weight message. This method targets the case of mobility, where the location of the leader needs to be changed, while the node hosting the previous leader is still functional. This light-weight method is possible because mobility, unlike failures, is triggered by user actions, and thus can be coordinated more efficiently. The basic idea behind Leader Handoff is to treat leadership as a logical role, rather than being physically attached to a single node. Then, a logical leader can move between nodes while maintaining its role without a Leader Election round.

4.2 Zone-Centric Quorums

DPaxos’ Zone-centric Quorums redefines Leader Election and Replication quorums with the goal of making Replication quorums as small as possible. This is due to the realization that the Replication phase is the most frequent phase in operation (a prolonged leader performs the Leader Election round only once and then skips it for future slots.) To this end, DPaxos utilizes a recent theoretical discovery that Paxos-based Replication and Leader Election quorums can be allocated arbitrarily with the following condition:

Definition 1. ***(Inter-Intersection Condition)*** *a Leader Election quorum must intersect with all Replication quorums [16].*

This is as opposed to the original requirement in the original Paxos protocol to have majority quorums for both Leader Election and Replication.

Zone-centric Quorums requires no modifications to the original Paxos protocol (Section 2) except on how quorums are defined. The new quorum definitions are no longer majority quorums, but rather ones that satisfy the inter-intersection condition (any Replication quorum, Q_r, must intersect with any Leader Election quorum, Q_{le}).

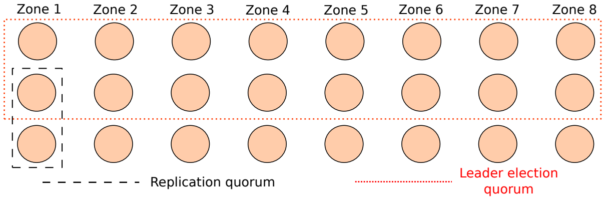


Figure 2: An example of a Zone-centric Leader Election and Replication quorums over the scenario of Figure 1.

The ideal Replication quorum, Q_r, is one with the smallest possible size. To tolerate failures, the smallest possible Replication quorum must include $f_d + 1$ nodes in $f_z + 1$ zones. Consider the topology in Figure 1 with 8 zones and assume that we want to tolerate one node failure ($f_d = 1$) and no zone failures ($f_z = 0$). In this case, a Replication quorum would be any pair of nodes in a zone. An example is shown in Figure 2 showing a Replication quorum consisting of two nodes in zone 1 (other Replication quorums are not shown). Now, a Leader Election quorum, Q_{le}, must intersect with *all* possible Replication quorums. Thus, it must span all zones because a Replication quorum is confined to a single zone. More generally, the Leader Election quorum must include $|Z| - f_z$ zones, where $|Z|$ is the number of zones. In each zone, the Leader Election quorum must cover enough nodes to ensure an intersection with any Replication quorums. This means it needs to include $|Z_i| - f_d$ nodes in zone i, where $|Z_i|$ is the number of nodes in zone Z_i. In Figure 2, all zones include 3 nodes, which means that the Leader Election quorum must include 2 nodes in each zone.

The inter-intersection condition suffices to ensure correctness because any node aspiring to be a leader (by running a Leader Election phase) will intersect, in the Leader Election phase, with the Replication quorums of the current leader and previous leaders. For example, consider a scenario on the topology in Figure 2. Figure 3 depicts the scenario where a node in zone 1 becomes a leader in slot i and then decides values from slots $i + 1$ to $i + 8$ while bypassing the Leader Election phase. Now assume that a node in zone 4 tries to become the leader in slot $i + 9$. It can only be a leader if it gets positive votes from a Leader Election quorum. A Leader Election quorum intersects with the Replication quorum in zone 1 in at least one node $\mathcal{A}$. Therefore, getting a positive vote from $\mathcal{A}$ will guarantee that the previous leader in zone 1 will not be able to propose a new value. The node in zone 4 now becomes a leader and bypasses the Leader Election phase for future slots until another node takes over the leader role.

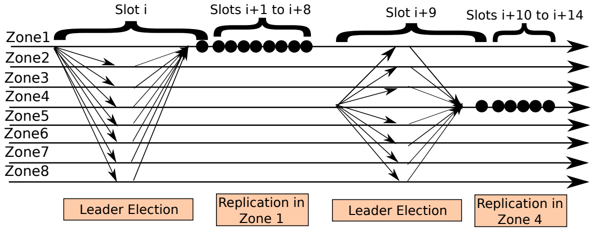


Figure 3: An example of a node in zone 4 taking over the role of a leader from a leader in zone 1 over the scenario in Figure 1. Given the large number of nodes, we represent nodes in a zone collectively with a single time line and communication within a zone is represented with a filled circle.

Summary. Zone-centric Quorums enables Replication quorums to be as small as possible and enables configuring fault-tolerance parameters, f_d and f_z. However, this results in expensive Leader Election quorums that must intersect with all Replication quorums. The rest of DPaxos’ design reduces the cost of Leader Election while maintaining small Replication quorums.

4.3 Expanding Quorums

DPaxos introduces a dynamic quorum allocation approach called *Expanding Quorums* to overcome the restrictive inter-intersection condition (Definition 1). Specifically, Expanding Quorums introduces small modifications to the Paxos protocol to allow smaller Leader Election quorums while maintaining as-small-as-possible Replication quorums. Optimizing the performance of the Leader Election phase is especially important for mobile and collaborative applications where users or workloads move frequently between physical locations. In such applications, the Leader Election phase is frequent as it occurs whenever the leader moves from one zone to another.

Expanding Quorums builds upon the following observation: Paxos ensures correctness by making an aspiring leader’s Leader Election quorum intersects with all possible Replication quorums that can be used by other leaders. However, *it suffices for a Leader Election quorum to only intersect with Replication quorums that are, or previously were, used by other leaders*—rather than *all* possible Replication quorums. Expanding Quorums utilizes this observation by making leaders announce the Replication quorums they are going to use. Aspiring leaders have to only intersect with the announced Replication quorums. The rest of this section describes how we generalize Paxos to distribute and react to such announcements.

Leader Election in Expanding Quorums has the additional tasks of (1) announcing the Replication quorums that are intended to be used (called *intents*), and (2) detecting and reacting to any intent announcements from other (aspiring) leaders. In Expanding Quorums,

Algorithm 1: The Leader Election phase in DPaxos Expanding Quorums (The Replication phase is identical to Paxos). Lines in red denote the changes compared to Paxos Leader Election.

```
Set $Q_{\mathcal{LE}}$ of leader election (LE) quorums
Set $Q_{\mathcal{R}}$ of arbitrary replication (R) quorums
// Any $Q_1, Q_2 \in Q_{\mathcal{LE}}$ must intersect
Leader Election phase (in: value $v$) {
  Pick proposal id $p$ larger than any known proposal
  Send prepare(p,intent) to a quorum $Q_{le} \in Q_{\mathcal{LE}}$
  if $Q_{le}$ responded with promise($q$, $v_q$, $p$,intents) then
    Send prepare(p,intent) to received intents
    if no nodes from any received intent responded with promise($q$, $v_q$, $p$, $intents$) then
    Terminate or retry
    if no $q$ and $v_q$ were received then
      $p' \leftarrow p$
      $v' \leftarrow v$
    else
      $p' \leftarrow$ highest received $q$ is promise()'s
      $v' \leftarrow$ value associated with $p'$
    Proceed to Replication phase (in: $p'$, $v'$)
  else
    Terminate or retry
}
```

Leader Election quorums must intersect with each other to ensure that an intent is detected by any future Leader Election phase. Therefore, Expanding Quorums *replaces* the inter-intersection condition with an *intra-intersection* condition for quorum allocation defined as the following:

Definition 2. ***(Intra-Intersection Condition)*** *any two Leader Election quorums must intersect.*

The Intra-intersection condition allows smaller Leader Election quorum allocation while maintaining small Replication quorum allocation as we will demonstrate by two incarnations of Expanding Quorums: Delegate and Leader Zone Quorums.

If an aspiring leader detects intents, then it must intersect with these intents by *expanding* the Leader Election quorum to intersect with at least one node in each intent's Replication quorum. Thus, rather than a static definition of Leader Election quorums, Expanding Quorums proposes a reactive, dynamic Leader Election quorum.

Expanding Quorums introduces the following changes to the traditional Paxos protocol to implement the announcement and detection of intents in addition to the expansion of Leader Election quorums (See Algorithm 1 for a summary of the changes to the Paxos algorithms):

- The prepare() message in the Leader Election phase includes a *Replication quorums intent* (or *intent* for short). The intent is the Replication quorum that the aspiring leader intends to use in case it advances to the Replication phase. It is possible to declare more than one intent in the same message.
- The promise() message in the Leader Election phase includes a list of previously received intents. Not included in the list are intents of unsuccessful prepare() messages that the node did not respond to positively with a promise.
- *Leader Election quorum expansion:* The Leader Election Quorum of an aspiring leader must intersect with the Replication quorums of every received intent. This is done by initiating a second round of communication to collect enough votes to intersect with the Replication quorums declared in the intents.

To summarize, Expanding Quorums tracks intents during the Leader Election phase and Leader Election quorums must expand to intersect with any declared intents. The only condition on the assignment of initial Leader Election quorums is to make them intersect with each other (Definition 2). Note that this is different from the original restriction where a Leader Election quorum must intersect with all Replication quorums (Definition 1). With Expanding Quorums, the allocation of Leader Election quorums is independent from Replication quorums, and thus, even the smallest of Replication quorums would not be limiting.

Next, We show how Expanding Quorums enables smaller Leader Election quorums with two strategies that we propose: Delegate Quorums (Section 4.3.1) and Leader Zone Quorums (Section 4.3.2). We discuss the safety of Expanding Quorums in Section 4.3.3. Then, we present the design of our intents' garbage collector in Section 4.3.4.

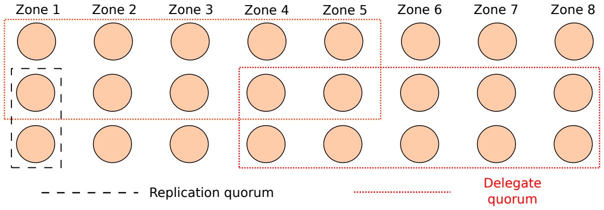


Figure 4: An example of Delegate Quorums over the scenario of Figure 1.

4.3.1 *Delegate Quorums.*

A *Delegate quorum* ($Q_{le} \in Q_{\mathcal{LE}}$) consists of nodes from a majority of zones. In every zone in the majority, a majority of nodes is part of the quorum. This ensures that any two Delegate Leader Election quorums intersect (Definition 2). Figure 4 shows an example of Delegate Quorums that only needs to communicate with five zones (This is in comparison to Flexible Paxos' Leader Election quorums that must intersect with *all* zones if replication quorums are confined to a single zone.) Note that the Delegate quorum does not intersect with all Replication quorums. However, it intersects with all other Delegate Quorums. Any intents received from these intersections during the Leader Election phase results in the aspiring leader expanding the Leader Election quorum to enforce intersection with the declared intents' Replication quorums.

Since the Delegate Quorums Leader Election quorum is an Expanding Quorum, intersection among Leader Election quorums is sufficient for correctness. To demonstrate this, consider the scenario in Figure 5. This scenario is over the topology in Figure 1 with eight zones and three nodes per zone, $f_d = 1$ and $f_z = 0$. Initially, a node in zone 1 becomes the leader in slot i by getting the votes from a majority of zones and then replicates within zone 1 until slot $i + 4$. Then, a node in zone 4 tries to become the leader in slot $i + 5$ by polling the votes from a majority of zones. This majority happens

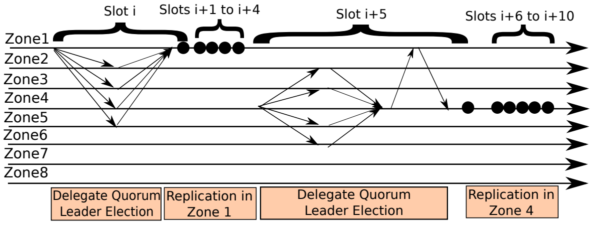


Figure 5: An example of a node in zone 4 taking over the role of a leader from a leader in zone 1 using a Delegate quorum over the scenario in Figure 1.

to not include zone 1. However, they intersect with the Delegate quorum that was started in slot i. Thus, they receive the intent of using a Replication quorum in zone 1. Therefore, the aspiring leader in zone 4 starts another round of communication to get the vote of nodes in zone 1. Only the vote of one node that intersects the intent's Replication quorum is sufficient. Afterwards, assuming that positive votes are collected, the node in zone 4 becomes a leader and starts deciding the values for future slots until a new leader is elected.

Note that the second round in Delegate Quorums is only needed if a Replication quorum intent is not covered in the first round. If there were no intents or the original majority intersected with all intents already, then there is no need to go to another round of a Delegate Quorums Leader Election. Also, it is possible to proactively collect votes of nodes from zones outside the majority during the first round to avoid the latency penalty of a second round. We will present more details about this optimization in Section 4.6.

An interesting case arises if the promises received at the second round of Delegate Quorums includes an intent that was not received in the first round. Although possible, the aspiring leader, A, can discard that intent. This is because such an intent is from a concurrent aspiring leader, B, that is guaranteed to receive the intents of A (since A did not receive the intents of B in A's first round, B is guaranteed to get the vote of one of A's voters *after* A).

4.3.2 Leader Zone Quorums. Leader Zone Quorums utilizes the Expanding Quorums technique to achieve small Leader Election quorums that are as small as a single zone (ideally, the zone hosting the current leader.) The main idea of a Leader Zone Quorum is to make all aspiring leaders contend to win the votes of a majority of nodes in a single zone that we call the *Leader Zone* ($\mathcal{Q}_{\mathcal{LE}}$ is the set of all majorities in the Leader Zone.) Because all aspiring leaders are contending to get the votes from the same zone, they all intersect with each other which satisfies the intra-intersection condition (Definition 2).

This definition of a Leader Zone is efficient if the current and aspiring leaders are close to each other, so that the Leader Zone can be chosen close to them. However, if the workload moves to a distant location, then a fixed Leader Zone will lead to expensive Leader Election, since aspiring leaders need to communicate with a distant Leader Zone. To rectify this, DPaxos allows changing the location of the Leader Zone. In the rest of this section, we begin by showing the additional changes to the Expanding Quorums algorithms to support Leader Zone quorums. Then, we show how the Leader Zone can migrate to follow the leader.

Algorithm 2: Leader Election phase in DPaxos Expanding Quorum with Leader Zone Quorums (Other routines are identical to ones in Algorithm 1). Lines colored in red denotes the changes compared to Algorithm 1.

```
Set $\mathcal{Q}_{\mathcal{LE}}$ initially to be all majority quorums in a select zone
Leader Election phase (in: value $v$) {
  Pick proposal id $p$ larger than any known proposal
  Send prepare(p,intent) to a quorum $Q_{le} \in \mathcal{Q}_{\mathcal{LE}}$
  **if** piggybacked information about a next Leader Zone $Z_i$ in transition is received **then**
    Set $\mathcal{Q}_{\mathcal{LE}}$ to contain any quorums $Q_{le}$ such that $Q_{le}$ is the union of two majority quorums one from the current Leader Zone and one from $Z_i$
    Send prepare(p,intent) to a quorum $Q_{le} \in \mathcal{Q}_{\mathcal{LE}}$
  **if** piggybacked information about a new completely transitioned Leader Zone $Z_i$ is received **then**
    Set $\mathcal{Q}_{\mathcal{LE}}$ to be the set of majority quorums in $Z_i$
    Send prepare(p,intent) to a quorum $Q_{le} \in \mathcal{Q}_{\mathcal{LE}}$
  **if** $Q_{le}$ responded with promise($q$, $v_q$, $p$, $intents$) **then**
    Send prepare(p,intent) to received intents
    **if** no nodes from any received intent responded with promise($q$, $v_q$, $p$, $intents$) **then**
      Terminate or retry
    **if** no $q$ and $v_q$ were received **then**
      $p' \leftarrow p$
      $v' \leftarrow v$
    **else**
      $p' \leftarrow$ highest received $q$ is promise()'s
      $v' \leftarrow$ value associated with $p'$
    Proceed to Replication phase (in: $p'$, $v'$)
  **else**
    Terminate or retry
}
```

The following summarizes the additions to Expanding Quorums to implement Leader Zone Quorums (see Algorithm 2):

- *Initial Leader Zone:* A zone is selected to be the initial Leader Zone. This zone can be selected arbitrarily but all nodes must agree on one zone to be the initial Leader Zone (*e.g.*, as part of the initial configuration of each node).
- *Leader Zone Quorums:* An aspiring leader performs Leader Election according to the Expanding Quorums algorithms in Algorithm 2, where $\mathcal{Q}_{\mathcal{LE}}$ is the set of all majority quorums of the Leader Zone. (It is possible to define Leader Zones to extend beyond a single zone if zone failures are to be tolerated. In such a case, a majority vote from the Leader Zones would be needed. For clarity of exposition, we focus on the case of a Leader Zone that is confined to a single zone.)
- *Leader Zone migration announcements:* If an announcement that the Leader Zone has changed its location, then the set of Leader Election quorums, $\mathcal{Q}_{\mathcal{LE}}$, is updated to reflect this change.

Now we present how the Leader Zone moves to another zone. This complicates the design as this makes $\mathcal{Q}_{\mathcal{LE}}$ dynamic and must be updated consistently to reflect the location of the new Leader Zone. The challenge with such movement is that all aspiring leaders must agree on the Leader Zone's location. If two leaders do not agree on which zone is the Leader Zone, then it is possible that they both get positive votes for their prepare() messages from two different zones. Such disagreements may happen due to failures and message delays, where announcements of the new Leader Zone's location are not yet propagated to all nodes.

Our solution to enable moving Leader Zone Quorums is a multi-step approach driven by any node i:

- *Step 1 (register a unique next Leader Zone):* node i registers zone Z_i as the next Leader Zone. Only one zone can be registered as the next Leader Zone. This is performed via a separate Paxos instance, that we call the Leader Zone Instance, run in the current Leader Zone, zone Z_j. (The Leader Zone instance is reconfigured to follow the location of the Leader Zone.) Node i successfully registers Z_i as the next Leader Zone by deciding the value Z_i in the log of the separate Paxos instance.
- *Step 2 (set up the transition phase):* In the transition phase, the next zone, Z_i, becomes part of the Leader Zone quorum, while the old Leader Zone, Z_j, still processes ongoing requests. This is performed by node i requesting that at least a majority of nodes in the current Leader Zone, Z_j, to: (1) send all the intents they have received so far back to i. Node i then ensures that a majority of nodes in Z_i maintains these intents, (2) piggyback information about the new Leader Zone, Z_i in their promise() messages, and (3) not add the intents received in future prepare() messages to their intents list.
During the transition phase, an aspiring leader that receives the piggybacked information about the next Leader Zone must receive promise() messages from two majorities, one from Z_i and another from Z_j.
- *Step 3 (Complete transition to the new Leader Zone):* To detach the old Leader Zone, Z_j, from the Leader Zone Quorums, we must ensure that any potential conflict will be resolved by consulting the new Leader Zone, Z_i, only. Completing step 2 ensures that all intents will be maintained by the new Leader Zone; past intents are maintained by explicitly requesting for them and new ones by making aspiring leaders send prepare() messages to Z_i. At this point, it is possible to announce to all nodes in all zones that Z_i is the new Leader Zone. This announcement can be lazily propagated in the background. An aspiring leader, that is not aware of the transition, will consult the old Leader Zone, Z_j, that will redirect it to the new Leader Zone, Z_i.

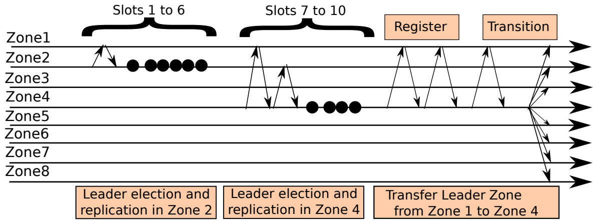

Slots 1 to 6
Slots 7 to 10
Register
Transition
Zone1
Zone2
Zone3
Zone4
Zone5
Zone6
Zone7
Zone8
Leader election and replication in Zone 2
Leader election and replication in Zone 4
Transfer Leader Zone from Zone 1 to Zone 4

Figure 6: An example of Leader Zone Quorums over the scenario of Figure 1.

An example of Leader Zone Quorums is shown in Figure 6 which is over the scenario in Figure 1 with eight zones, $f_d = 1$, and $f_z = 0$. Initially, Zone 1 is the Leader Zone. Consider that a node i in Zone 2 wants to become a leader. It sends prepare() messages to a majority of nodes in Zone 1 with the intent to use a Replication quorum in Zone 2. A majority of nodes in Zone 1 replies with promise() messages to node i. There are no previous intents in this case. Therefore, node i proceeds to the Replication phase with a Replication quorum in Zone 2 and decides values for slots 1 to 6. Afterwards, node j in Zone 4 attempts to become a leader by sending prepare() messages to a majority of nodes in the Leader Zone, Zone 1. The majority in Zone 1 responds with promise() messages that include the intent of node i (a Replication quorum in Zone 2). Therefore, the Leader Election quorum expands to include the Replication quorums in Zone 2. Node j sends prepare() messages to nodes in Zone 2. Then, enough nodes to intersect with node i's intent in Zone 2 respond with promise() messages. At that point, node j proceeds to the Replication phase and decides the values in slots 7 to 10. After slot 10 is decided, node j decides to transfer the Leader Zone to Zone 4. First, node j registers that Zone 4 is the next Leader Zone. It does so by deciding the value of "Zone 4" in the separate Paxos process in Zone 1 that is designated for this purpose. This entails going through the Leader Election and Replication phases of the separate Paxos instance, shown in the figure as two rounds of communication between Zone 4 and Zone 1. Then, node j sends requests to a majority of nodes in Zone 1 to begin the transition phase (intents are sent from Zone 1 to Zone 4, information about the transition are piggybacked in promise() messages from Zone 1, and new incoming intents are not maintained in Zone 1.) During this time, aspiring leaders (not shown in the figure) would have to get the votes from majorities in both Zone 1 and Zone 4. They would know about being in the transition phase by receiving the piggybacked information from Zone 1's promise() messages. To complete the transition, node j sends announcement messages to all nodes, that declares Zone 4 to be the new Leader Zone. At this point, aspiring leaders (not shown) only need to get a majority of votes from Zone 4 in the leader election phase. Aspiring leaders that are not aware of the transition (*e.g.*, due to dropped or delayed announcements) would send their prepare() messages to Zone 1, that in turn notifies them of the new Leader Zone.

4.3.3 *Safety.*

The safety of a consensus protocol is a guarantee of *consistency* (at most one value can be decided) and *non-triviality* (only a proposed value can be decided). In this section, we build upon the proof of Flexible Paxos' safety [16] to prove the safety of DPaxos.

Flexible Paxos [16] shows that both safety properties are satisfied for Flexible Paxos by proving the following theorem

Theorem 1. *If a value v with proposal id p is decided, then any message propose(p_2,v_2) where $p_2 > p$ satisfies $v = v_2$.*[1]

The proof in Flexible Paxos relies on the intersection condition between the Replication quorum of proposal p, Q_r^p, and the Leader Election quorum of proposal p_2, $Q_{le}^{p_2}$. This is true by definition for Flexible Paxos (the inter-intersection condition). However, DPaxos' Replication and Leader Election quorums do not intersect

[1] This is an adaptation of Theorem 2 in Flexible Paxos [16]

by definition (the intra-intersection condition). DPaxos enforces the intersection between Q_r^p and $Q_{le}^{p_2}$ by expanding the Leader Election quorum, $Q_{le}^{p_2}$. To avoid confusion we introduce two notations: we use $Q_{ole}^{p_2}$ to denote the original Leader Election quorum before expansion and use $Q_{ele}^{p_2}$ to denote the Leader Election quorum after expansion. Now, we show that DPaxos satisfies the intersection condition between Q_r^p and $Q_{ele}^{p_2}$:

Theorem 2. *Consider a Replication quorum with proposal id p, Q_r^p, and a Leader Election quorum with proposal id p_2, where $p < p_2$. DPaxos ensures that $Q_r^p \cap Q_{ele}^{p_2} \neq \phi$.*

Proof. This is a proof by contradiction. Assume to the contrary that both quorums are used and that $Q_r^p \cap Q_{ele}^{p_2} = \phi$. By definition, any two Leader Election quorums intersect. Therefore, the Leader Election quorum in p and the Leader Election quorum in $p2$ intersect ($Q_{ole}^{p} \cap Q_{ole}^{p_2} \neq \phi$). There is at least one node $\mathcal{A}$ in this intersection. There are two possible cases:

- Case 1 ($\mathcal{A}$ received prepare(p,intent=Q_r^p) before prepare(p_2,intent=$Q_r^{p_2}$)): in this case, $\mathcal{A}$ responds to prepare(p_2,intent=$Q_r^{p_2}$) with the intent Q_r^p. This triggers a quorum expansion of $Q_{ole}^{p_2}$ to $Q_{ele}^{p_2}$ that intersects with Q_r^p, which is a contradiction.
- Case 2 ($\mathcal{A}$ received prepare(p_2,intent=$Q_r^{p_2}$) before prepare(p,intent=Q_r^p)): In this case, $\mathcal{A}$ does not respond to prepare(p,intent=Q_r^p) because $p < p_2$. Therefore, the Leader Election with proposal id p fails and the Replication quorum Q_r^p is not used, which is a contradiction.

□

Therefore, DPaxos ensures that $Q_r^p \cap Q_{ele}^{p_2} \neq \phi$. This suffices to show safety by referring to Flexible Paxos' proof that safety is guaranteed with the condition that $Q_r^p \cap Q_{le}^{p_2} \neq \phi$ [16], where $Q_{ele}^{p_2}$ represents the Leader Election quorum of proposal p_2.

4.3.4 *Intents Garbage Collection.*

With continued operation and Leader Election rounds, the intents from aspiring leaders accumulate at nodes. This can have serious performance ramifications. The accumulation of many intents means that future leaders need to intersect with a larger number of nodes in a larger number of zones. Also, a large number of intents increases the size of promise messages—increasing the communication overhead.

DPaxos employs a garbage collection process that gradually removes *obsolete intents* that future leaders do not need to intersect with. Any intent is subject to garbage collection after its corresponding leader loses leadership. This includes intents of failed leader election attempts (where a majority of promise messages were not sent back to the aspiring leader) as well as intents of successful leader election attempts with potentially subsequent successful replication rounds but who have subsequently lost their leadership to a successful new leader.

The garbage collector is a separate process that performs two tasks: (1) determine the set of obsolete intents by polling information from acceptors, and (2) remove obsolete intents from acceptors. More than one garbage collector can co-exist, and garbage collectors could be shutdown and resumed arbitrarily. Algorithm 3 shows the procedure followed by a garbage collection process. First, the garbage collector picks a node i arbitrarily. In DPaxos, we pick nodes in a round-robin. The garbage collector polls the following information from node i: *the largest proposal id that node i received with a propose message*. We will denote this value as $\mathcal{P}_i$. (It is important to note that this value depends on the received propose messages and not the prepare messages.) The garbage collector maintains the maximum $\mathcal{P}_i$ value and denote it as $\mathcal{P}$. The garbage collector will consider any intent as obsolete if its proposal number *p is smaller than* $\mathcal{P}$. The garbage collector asynchronously broadcasts the new $\mathcal{P}$ value to all nodes. A node that receives the new $\mathcal{P}$ value removes all intents with proposal numbers *lower than* $\mathcal{P}$. The node removes the intent whether it belongs to a failed leader election attempt or a successful one.

Algorithm 3: The algorithm performed by a garbage collection process

```
$\mathcal{P}$ := the garbage collection threshold, initially 0
Garbage Collection {
  Repeat {
    i ← select a node arbitrarily
    $\mathcal{P}_i$ ← poll the largest proposal number that node $i$ accepted
    **if** $\mathcal{P}_i > \mathcal{P}$ **then**
      $\mathcal{P} = \mathcal{P}_i$
      Propagate $\mathcal{P}$ to all acceptors asynchronously
  }
}
```

Central to the correctness of the garbage collection algorithm is that an intent with a proposal number lower than $\mathcal{P}$ is an obsolete intent. To ensure correctness, we prove the following:

Theorem 3. *An intent's replication quorum cannot accept any new propose messages with a proposal number p, where p is the proposal number corresponding to the intent and $p < \mathcal{P}$.*

Proof. The proof is presented in Section C. □

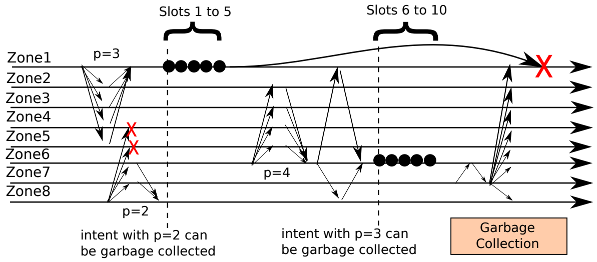


Figure 7: An example of garbage collecting obsolete intents.

Figure 7 shows an example of garbage collecting obsolete intents. There are 8 zones, and assume that the Delegate Expanding Quorums is used. (A Leader Election quorum consists of a majority of zones.) Initially, a node, z_1, in Zone 1 performs a successful Leader Election phase with proposal id 3. Concurrently, another node, z_8, in Zone 8 performs a Leader Election phase with proposal id 2. However, z_8's proposal id is lower than 3 and Zones 4 and 5 already responded to z_1's prepare messages. This causes the Leader Election to fail due to not getting the votes of a majority of zones.

At this point, Zones 1 to 5 have the intent of z_1, that we will call I_1, and Zones 6 to 8 have the intent of z_8, that we will call I_8. Later, node z_1 starts deciding values for slots 1 to 5 via local Replication phases. As soon as z_1's propose messages start to arrive to nodes in Zone 1, a garbage collector that polls nodes in Zone 1 will update its $\mathcal{P}$ value to 3. And thus, a garbage collector from that point can garbage collect the intents of I_8. However, assume for now that the garbage collector is not active yet. After deciding a value for slot 5, the propose messages for slot 6 are delayed (represented as the curved line in Zone 1). While the propose messages are delayed, a node, z_6, in Zone 6 performs a Leader Election phase with proposal id 4 and Leader Election quorum consisting of Zones 2 to 6. Because it is the highest proposal id so far, Zones 2 to 6 reply with promise messages. However, the promise messages from Zones 2 to 5 include the intent I_1 and the promise messages from Zone 6 include the intent I_8. Node z_6 expands the Leader Election quorum to include zones 1 and 8 and completes the Leader Election phase. Then, z_6 starts committing values locally in Zone 6 for slots 6 to 10. At that point a garbage collector polls nodes in Zone 6 for their state and concludes that the value for $\mathcal{P}$ is 4. It asynchronously broadcasts the value to all nodes. When a node receives the new $\mathcal{P}$ value, it removes all intents with lower proposal ids, which are I_1 and I_8 in this case.

The delayed in-flight propose message from z_1 that tried to decide the value for slot 6 eventually arrives to the replication quorum in Zone 1 after the intent I_1 has been garbage collected. However, they are not accepted because at least one node in the Replication quorum must have participated in z_6's Leader Election quorum. Therefore, even if a node now becomes a new leader without consulting with the Replication quorum in Zone 1, it is guaranteed that the Replication quorum in Zone 1 is not going to accept proposals with proposal id 3. More generally, an intent is only garbage collected after at least one node in its Replication quorum promises not to accept proposals from the intent's leader.

We would like to note that it is possible to perform garbage collection of intents in various ways. We believe that our method strikes a good balance between efficiency and simplicity. However, we briefly present other methods that may be more suitable for system environments with varying requirements and goals. One possibility is to leverage the methods proposed in Stoppable Paxos [26] that would allow garbage collecting of all intents at predetermined points when the Paxos instance "stops." This is especially desirable in cases where a one-shot, periodic garbage collection process is more desirable than DPaxos' continuous garbage collection that may add overhead to concurrent computation. We discuss Stoppable Paxos in more details in Section B.1. Also, DPaxos' garbage collection methods can be adopted more aggressively. For example, we can discover an obsolete intent as soon as a node in its Replication quorum has sent a promise with a higher proposal number. Another optimization relies on that a newly elected leader with proposal id p knows that all intents with lower proposal ids are obsolete (after completing its Leader Election phase). Therefore, it can broadcast to all nodes that the new value of $\mathcal{P}$ is p. Garbage collection can also be done independently at every node. When a node receives a propose message with proposal id p, then it can garbage collect all the internal intents with lower proposal ids.

Garbage collection can be affected by node and communication failures. Because it relies on polling information from nodes, a disruption of communication may delay garbage collection. However, the garbage collection process inherits its liveness properties from Paxos. This is because the successful completion of a Leader Election phase is equivalent to the ability to garbage collect all previous intents.

4.4 Leader Handoff

DPaxos introduces a light-weight mechanism to change the location of the leader without invoking a Leader Election phase. The observation that led to this technique is that there are two motivations to elect a new leader: (1) Fault-tolerance: the current leader has failed, and (2) Mobility: the users or workload have moved and we want to move the location of the leader. Leader Election in Paxos works to serve both purposes. However, we have found that for mobility, changing the location of the leader may be performed without going through the Leader Election phase by exploiting that the current leader can participate in the process. We call this technique, *Leader Handoff*. This technique can be applied to Paxos variants in general and is not restricted to DPaxos.

The basic idea is to treat the leader role as a logical role rather than being physically tied to a physical node. In principle, a leader can use any Replication quorum. Additionally, there is no requirement for the leader to be in the same location all the time. The Leader Handoff technique enables a leader to *relinquish* its privileges as a leader to another node. This can be via a simple relinquish() message from the current leader to the new leader consisting of the current state of the leader and the slots that the leader would like to relinquish (the set of relinquished slots may be unbounded). A leader can send this message only once for any slot. Also, after sending the message, it refrains from acting as a leader for these slots.

If the message from the old leader to the new leader is lost, then neither of them can act as the leader. Rather, a Leader Election phase must take place. An additional restriction if this is used in conjunction with Expanding Quorums is that the new leader can only use Replication quorums that were declared by the intent of the leader that relinquished the leadership (more about declaring multiple intents in Section 4.6.)

4.5 Read Leases

To optimize the performance of read-only operations and transactions, many Paxos variants utilize a *read lease* [5, 9, 10, 14, 33, 34]. A read lease allows a lease holder (or holders [34]) to respond to a read-only request independently without making the read request commit as a command that interferes with other commands and read-write transactions. The lease has a duration. Granting a lease to a node (or group of nodes) is a guarantee that no other node will be able to hold a read lease and that all writes are channeled through the lease holders until the read lease expires (10s is an example of a lease duration in real-world scenarios [10].) These two guarantees ensure that the lease holder has the most recent copy of data during the lease duration.

Read leases have been used in various ways in Paxos-based systems. Megastore [5] enforces writes to synchronously coordinate with all replicas, enabling each replica to act as if it has a read lease. Others, such as Spanner [10], EPaxos [33], and Chubby [9], utilizes the Multi-Paxos leader to act as a lease holder. Moraru et. al. [34] propose a quorum-based read lease, where a select group of replicas act as lease holders. DPaxos targets applications with spatial locality workloads, making the leader-based read lease the more appropriate alternative. DPaxos utilizes a leader-based read lease approach, however, other approaches can also be adapted to DPaxos as well. In leader-based read leases, a multi-Paxos leader sends requests for lease votes. If a majority responds with lease votes, then the leader has the lease throughout its duration. Leases are then extended either implicitly via a new successful decided value (where accept messages act as lease votes as well) or explicitly by new lease vote requests.

DPaxos has the special property that, unlike other systems with leader-based read leases, its quorum sizes can be less than a majority and getting votes from a Replication quorum is more efficient than a Leader Election quorum. This invites a careful consideration of DPaxos read leases to ensure the practicality and safety (*i.e.*, linearizable reads) of leases in DPaxos. In terms of safety, DPaxos needs to ensure that no two nodes think they hold read leases concurrently. This can be ensured by getting lease votes from a majority of nodes. This, however, is expensive. Alternatively, DPaxos implements read leases with the objective of restricting the communication needed to renew a lease to a single Replication quorum. DPaxos implements read leases in the following way that is inspired from the *master lease* approach [9]: (1) Lease requests and votes can only be implicit by piggybacking lease requests with propose messages and piggybacking lease votes with accept messages, (2) A lease vote (accept message) has the implicit promise not to participate in Leader Election (*i.e*, not respond to prepare messages) until the lease expires. With this method, a leader can acquire or renew a read lease with votes from its Replication quorum only. Safety is preserved because no node can get a promise message from the current leader's Replication quorum (and thus cannot become a leader) before the lease expires. Also, garbage collection does not threatens the integrity of leases. Because no node can be elected a leader before the lease expires, the intent corresponding to the current lease holder cannot be garbage collected.

We present an experimental evaluation of the use of master leases to serve read-only requests (Section A.2).

4.6 Summary and Practical Considerations

Summary. The following points summarize the characteristics and trade-offs of DPaxos' quorum allocations and methods:

- A DPaxos *Replication* quorum is any collection of f_d + 1 nodes in f_z + 1 zones.
- A Flexible Paxos (and the Zone-centric quorum without Expanding Quorums) *Leader Election* quorum consists of enough nodes to intersect with all Replication quorums. This equals to all zones minus f_z. In each zone, the Leader Election quorum includes all nodes minus f_d.
- A *Delegate* quorum is an Expanding Quorum that consists of nodes from a majority of zones and a majority of nodes from each of these zones.
- A *Leader Zone Quorum* Leader Election quorum is an Expanding Quorum that consists of a majority of nodes in the Leader Zone or Zones (during transition, Leader Election may span more zones than the normal case.)
- *Leader Handoff* enables a leader to relinquish all or part of the slots where it is a leader via a single light-weight message. However, if the leader fails, a new leader can only be elected via a Leader Election round.

Configuration. In DPaxos, we present various methods to allocate quorums, such as Delegate Quorums, Leader Zone Quorums, and Leader Handoff. These methods have different performance characteristics and which method would perform better depends on the environment. We show how different environments cause different outcomes of DPaxos methods in the experimental evaluation (Section 5). Additionally, DPaxos relies on the correct assessment and prediction of the workload's spatial locality characteristics. In this paper, we do not tackle the problem of configuration. This is an added complexity that system administrators will need to handle in comparison to using some other Paxos variants. However, it is possible to adapt automatic allocation and configuration methods for geo-replicated data stores [49, 50] to aid in the configuration of DPaxos. We believe this has a promising potential as an avenue of future research.

Use of multiple intents. In Expanding Quorums, we discussed how an aspiring leader must send an intent in the prepare() message. This can be generalized to many intents per aspiring leader. This is useful to ameliorate the restriction of only using the declared intents. Consider a leader that declares two intents rather than one. This leader has the flexibility of choosing either intent, in case one of them becomes slow or inaccessible. Otherwise, with a single declared intent, changing the Replication quorum would require a Leader Election round. The drawback of declaring more than one intent is that the intersection requirement for future aspiring leaders becomes bigger (they need to intersect with every declared intent.) This trade-off between the number of declared intents and the intersection requirement of future leaders depends on the workload. The strategy of allocating intents is left as a design decision.

Consolidate multiple rounds into a single round. In Expanding Quorum, an aspiring leader goes through a second round if nodes responded with intents. It is possible to consolidate the first Leader Election round with the second Leader Election round (to intersect with Replication Quorums in intents). For example, consider an aspiring leader L that is getting the votes from a Leader Election quorum, Q_{le}. Assume that node L received intents to intersect with a Replication quorum Q_i. In our description, node L will have two rounds of communication: the first with Q_{le} and the second with Q_i. However, the messages that are sent in both rounds are identical. Furthermore, sending prepare() messages to arbitrary nodes is permissible. Thus, it is possible for L to have sent the prepare() messages to both Q_{le} and Q_i simultaneously, thus consolidating the two rounds into one. The challenge in such consolidation is knowing whether to send the prepare() messages to nodes other than Q_{le}. In our scenario, L might not have known that Q_i would be in the intents list of Q_{le}. Such consolidation is only applicable when there are enough information to predict the intents before receiving them.

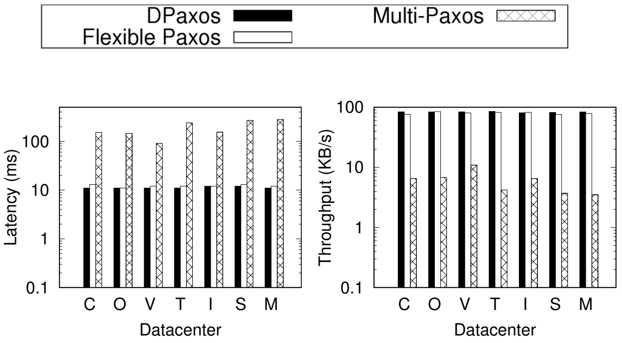


Figure 8: The performance of DPaxos in a scenario with seven datacenters

5 EVALUATION

In this section, DPaxos is evaluated on a real deployment on seven Amazon AWS datacenters, where each datacenter represents a zone.

The seven Amazon AWS datacenters we use are at California (*C*), Virginia (*V*), Oregon (*O*), Tokyo (*T*), Ireland (*I*), Singapore (*S*), and Mumbai (*M*). The Round-Trip Time (RTT) latency between each pair of datacenters is shown in Table 1 in the appendix. There are three nodes in each datacenter. We emulate the case that these nodes are placed on different edge locations by adding an artificial communication delay (10ms) between them. The delay only affects the communication between the nodes that are modeled to be in the same zone but on different edge nodes. The used machines (named m4.large) runs Linux and have two virtualized CPUs and 8 GB memory.

We use a simple workload consisting of small OLTP transactions. Each transaction has five operations (selected randomly from one million keys) and each operation has a value size of 50 Bytes. Half the operations are reads and the other half are writes. All transactions are read-write transactions. We present experiments with read-only transactions in Section A.2. Throughout this section, DPaxos will be evaluated with the following fault-tolerance level: $f_d = 1$ and $f_z = 0$, which tolerates a single datacenter failures. Each experiment runs for 1 minute. (running experiments for longer durations did not result in any significant difference.)

We compare DPaxos with Multi-Paxos, Flexible Paxos [16], and leaderless Paxos. There are a number of leaderless Paxos variants, where the Leader Election round can be bypassed, such as Egalitarian Paxos [33], Fast Paxos [24], and MDCC [18]. We compare with leaderless Paxos variants by assuming the optimal case of a Replication Quorum consisting of a majority of nodes. Such a quorum size may lead to inconsistency, but nonetheless would provide a benchmark of the best-case performance of leaderless Paxos variants.

5.1 Replication Phase Performance

We begin by presenting our evaluation of the Replication phase's performance. The Replication phase is especially important for prolonged leaders, where the Leader Election phase is bypassed. To evaluate the Replication phase, we measure the latency and throughput of the Replication phase in each one of the seven datacenters. This emulates the case of seven partitions, where each partition is located and accessed in one of the seven datacenters. A prolonged leader is located in the same zone as its partition. In this experiment, prolonged leaders decide 1 KB batches of transactions, and bypass the Leader Election phase. The results are shown in Figure 8, comparing DPaxos with Flexible Paxos and Multi-Paxos. Both DPaxos and Flexible Paxos decide values with an average latency of 11−13ms in all locations. This is expected as they have identical Replication Quorums. However, Multi-Paxos latency varies between 91ms for Virginia and 282ms for Mumbai. Multi-Paxos pull the votes of the majority, and thus the location of the proposer affects the observed latency. Throughput results are presented as the number of Bytes committed in a second. Throughput numbers reflect the effect of the latency difference between DPaxos and Flexible Paxos on one hand, and Multi-Paxos on the other hand. DPaxos and Flexible Paxos achieve similar throughput ranging from 75.8 KB/s to 85.2 KB/s. Multi-Paxos' throughput varies based on the location, ranging from 3.5 KB/s in Mumbai (the datacenter with the highest latency) up to 10.9 KB/s in Virginia (the datacenter with the lowest latency). Overall, the average throughput of DPaxos and Flexible Paxos is 23× the average throughput of Multi-Paxos.

The cause for the performance difference is that DPaxos and Flexible Paxos' Replication phase performance is independent of the nodes outside of the zone. However, Multi-Paxos' performance depends on the topology and how far are datacenters from each other. As the deployment covers a larger geographical region, and a larger number of nodes, as the majority vote becomes more expensive.

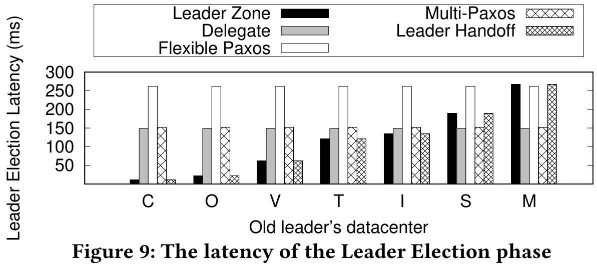
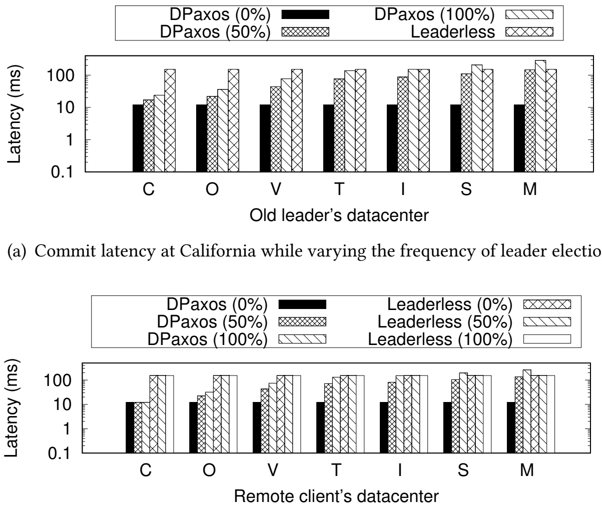


Figure 9: The latency of the Leader Election phase

5.2 Leader Election Performance

In this section, we evaluate the performance of the Leader Election round. We compare the two DPaxos Leader Election variants (Leader Zone and Delegate) with Flexible Paxos, Multi-Paxos, and Leader Handoff. The results are shown in Figure 9. In this experiment, we measure the Leader Election latency as observed by an aspiring leader in California. The location of the previous leader may affect the Leader Election latency (*e.g.*, for Leader Zone Quorum) and thus we vary, in the x-axis, the location of the previous leader. DPaxos Leader Zone Leader Election is affected by the location of the Leader Zone. In this set of experiments, we assume that prior Leader Election attempts have been garbage collected, and thus there are no intents other than the previous leader's intent and the Leader Zone has moved to the previous leader's zone. (In Section A.4 we show the effect of Leader Election while garbage collection of intents is still in progress.) The Leader Zone Leader Election takes one round to the previous leader's zone and ranges



(a) Commit latency at California while varying the frequency of leader election

(b) Commit latency while varying the amount of remote requests.

Figure 10: Comparing DPaxos with an optimal representation of leaderless Paxos variants

from 11ms when the previous leader is in the same zone to 267ms when the leader is in Mumbai. DPaxos Delegate Quorums and Multi-Paxos Leader Election takes a round to the closest zones to California, irrespective of the location of the previous leader. The Leader Election round takes 149–152ms for Delegate Quorums and Multi-Paxos Leader Election. Compared with DPaxos Leader Zone, Delegate Quorums and Multi-Paxos are faster if the previous leader is either in Singapore or Mumbai. Flexible Paxos Leader Election is the most expensive, requiring collecting votes from all zones. In this case, the latency to get the votes from all zones is 262ms (equivalent to the RTT between California and Mumbai). Leader Zone Quorums achieves a similar high latency only when the previous leader is in Mumbai.

Leader Handoff is our light-weight technique to relinquish leadership from an old leader to a new leader. Leader Handoff has similar performance characteristics to the Leader Zone Quorum. However, Leader Handoff can only be performed with the cooperation of the previous leader.

5.3 Comparing with leaderless Paxos

Leaderless Paxos variants bypass the Leader Election phase which offer two performance advantages: (1) Even in the case of failures, there is no need to invoke Leader Election, (2) Multiple proposers (at different locations) can decide slots simultaneously. Figure 10 shows the results of two experiments comparing with leaderless Paxos. In both experiments, we measure the performance of a proposer in California. The first experiment (Figure 10(a)) evaluates the overhead of the Leader Election phase of DPaxos in comparison to the performance of a leaderless Paxos that does not require Leader Election. The x-axis is the location of the previous leader. The y-axis is the decision latency (the Replication phase latency in addition to the Leader Election latency if invoked). For DPaxos, we vary the percentage of requests that invoke Leader Election, and show the 0% case (emulating the case of a prolonged leader that did not fail or move), the 50% case (emulating the case when Leader Election is invoked in every other request) and the 100% case (emulating the case when Leader Election is invoked in every request). In the 0% case, DPaxos' latency is 12ms, which is the Replication phase latency. In the 50%, the Leader Election overhead causes the average latency to range between 17ms (when the previous leader is in California but in another node) and 147ms (when the previous leader is in Mumbai). In the 100% case, the latency ranges between 24ms and 286ms, experiencing the full overhead of Leader Election. The optimal leaderless Paxos latency is 152ms. This means that even in the 50% case, DPaxos outperforms leaderless Paxos regardless of the location of the previous leader. In the 100% case, leaderless Paxos outperforms DPaxos only if the previous datacenter is in Singapore or Mumbai. Although the 50% and 100% Leader Election rate are extreme and will be far from a typical workload, DPaxos still manages to outperform the leaderless variant due to its low Replication latency.

The second experiment (Figure 10(b)) evaluates DPaxos with leaderless Paxos when there are remote requests (ones that did not originate in the same zone as the partition leader). In this set of experiments, the leader is in California and a subset of the users (either 0%, 50%, or 100%) are in another zone. DPaxos performs best for the requests that originates at California. Therefore, the best latency (12ms) is achieved when there are no remote requests (0% in the figure). In the 50% and 100% cases, half or all the requests originates at another datacenter (the source of the remote requests is in the x-axis). The remote requests are first forwarded to the leader in California, which in turn processes them and then replies back the decision to their clients. The effect depends on how far the client is from the leader (California). The farthest datacenter from California is Mumbai, where a remote request's latency is 260ms. For all other cases, the latency ranges between 22ms and 195ms. Leaderless Paxos' latency is if all requests originates in California is 152ms. In the 50% case, the latency ranges from 122ms to 217ms, and in the 100% case, the latency ranges from 91ms to 282ms. The only case where leaderless Paxos achieves a lower latency than DPaxos is in the 100% case where the remote requests originate in Mumbai. For the other cases, the latency gap shrinks. This shows that remote requests do indeed affect DPaxos' magnitude of improvement compared to leaderless Paxos variants. Also, leaderless Paxos can outperform DPaxos for workloads where there is little or no locality of access.

We present additional experimental evaluations in Section A.

6 CONCLUSION

In this paper, we propose DPaxos, a Paxos-based protocol for data management on the edge. We show that designing specifically for the new edge environment yields significant performance rewards. DPaxos includes three main proposals: (1) Zone-centric Quorums that utilizes Flexible Paxos [16] to make Replication quorums small and close to users, (2) Expanding Quorums that enables both small Replication and Leader Election quorums that grow dynamically in the presence of conflicts, and (3) Leader Handoff that targets supporting mobility by a light-weight method of relinquishing leadership. These proposals improve performance significantly as we show on a real deployment across 7 datacenters.

7 ACKNOWLEDGEMENTS

This work is supported by NSF grant CSR 1703560.

REFERENCES

[1] D. Agrawal and A. El Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *VLDB*, volume 90, pages 243–254, 1990.

[2] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. Multileader wan paxos: Ruling the archipelago with fast consensus. *arXiv preprint arXiv:1703.08905*, 2017.

[3] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 577–588. ACM, 2013.

[4] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[5] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. Innovative Data Systems Research*, pages 223–234, 2011.

[6] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 111–120. IEEE, 2012.

[7] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[8] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[9] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proc. 26th ACM Symp. Principles of Distributed Computing*, pages 398–407, 2007.

[10] J. Corbett et al. Spanner: Google's globally-distributed database. *OSDI*, 2012.

[11] I. Eyal, K. Birman, and R. van Renesse. Cache serializability: Reducing inconsistency in edge transactions. In *ICDCS*, pages 686–695. IEEE, 2015.

[12] D. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[13] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[14] C. Gray and D. Cheriton. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*, volume 23. ACM, 1989.

[15] H. S. Gunawi, A. Laksono, R. O. Suminto, M. Hao, J. Adityatama, K. J. Eliazar, and A. D. Satria. Why does the cloud stop computing? lessons from hundreds of service outages. *SoCC*, 2016.

[16] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696*, 2016.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. 2010 USENIX conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[18] T. Kraska et al. Mdcc: Multi-data center consistency. In *EuroSys*, pages 113–126, 2013.

[19] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, 1991.

[20] A. Kumar and S. Y. Cheung. A high availability $\sqrt{n}$ hierarchical grid algorithm for replicated data. *Information Processing Letters*, 40(6):311–316, 1991.

[21] L. Lamport. The part-time parliament. *ACM Trans. Computer Systems*, 16(2):133–169, May 1998.

[22] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[23] L. Lamport. Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

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

[25] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313. ACM, 2009.

[26] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

[27] L. Lamport and M. Massa. Cheap paxos. In *Dependable Systems and Networks, 2004 International Conference on*, pages 307–314. IEEE, 2004.

[28] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *OSDI*, pages 467–483, 2016.

[29] Y. Lin et al. Enhancing edge computing with database replication. In *SRDS*, pages 45–54, 2007.

[30] M. Maekawa. A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems (TOCS)*, 3(2):145–159, 1985.

[31] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commits. *VLDB*, 6(9), 2013.

[32] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *OSDI*, volume 8, pages 369–384, 2008.

[33] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[34] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[35] F. Nawab, D. Agrawal, and A. El Abbadi. Message futures: Fast commitment of transactions in multi-datacenter environments. In *CIDR*, 2013.

[36] F. Nawab, D. Agrawal, and A. El Abbadi. Nomadic datacenters at the network edge: Data management challenges for the cloud with mobile infrastructure. In *EDBT*, pages 497–500, 2018.

[37] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *SIGMOD*, pages 1279–1294, 2015.

[38] F. Nawab, V. Arora, V. Zakhary, D. Agrawal, and A. El Abbadi. A system infrastructure for strongly consistent transactions on globally-replicated data. *IEEE Data Eng. Bull.*, 40(4):3–14, 2017.

[39] F. Nawab et al. Chariots: A scalable shared log for data management in multi-datacenter cloud environments. In *EDBT*, pages 13–24, 2015.

[40] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. Planet: making progress with commit processing in unpredictable environments. In *SIGMOD*, pages 3–14, 2014.

[41] S. Patterson, A. J. Elmore, F. Nawab, D. Agrawal, and A. El Abbadi. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. *PVLDB*, 5(11):1459–1470, 2012.

[42] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1311–1326. ACM, 2015.

[43] H. Saxena and K. Salem. Edgex: Edge replication for web applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 1041–1044, 2015.

[44] A. Singla, B. Chandrasekaran, P. Godfrey, and B. Maggs. The internet at the speed of light. In *HotNets*. ACM, 2014.

[45] A. Thomson and D. J. Abadi. Calvinfs: consistent wan replication and scalable metadata management for distributed file systems. In *FAST*, 2015.

[46] A. Thomson et al. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[47] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[48] W. Wei, H. T. Gao, F. Xu, and Q. Li. Fast mencius: Mencius with low commit latency. In *INFOCOM, 2013 Proceedings IEEE*, pages 881–889. IEEE, 2013.

[49] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi. Db-risk: The game of global database placement. In *SIGMOD*, 2016.

[50] V. Zakhary, F. Nawab, D. Agrawal, and A. El Abbadi. Global-scale placement of transactional data stores. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.*, pages 385–396, 2018.

[51] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, pages 276–291, 2013.

A ADDITIONAL EXPERIMENTS

A.1 Batching

Batching has been commonly used in Paxos-based protocols to utilize Multi-Paxos slots more efficiently. This is especially critical for multi-datacenter environments where each round takes a large amount of time relative to the speed of computation. Increasing the batch size results in better utilization of the replication round and thus increases throughput. Figure 11 shows experiment results that quantify the effect of the batch size on throughput. We vary the batch size from 1 KB to 100 KB. Increasing the batch size from 1 KB to 100 KB leads to increasing throughput by a factor of 68× for DPaxos, 64× for Flexible Paxos and 25× for Multi-Paxos. Most of the throughput improvement is achieved when increasing the batch size to 50 KB. Increasing the batch size beyond 50 KB yield relatively lower throughput improvement for DPaxos and Flexible Paxos (an improvement factor of 1.5× for both). For Multi-Paxos, increasing the batch size beyond 50 KB decreases throughput, signaling that

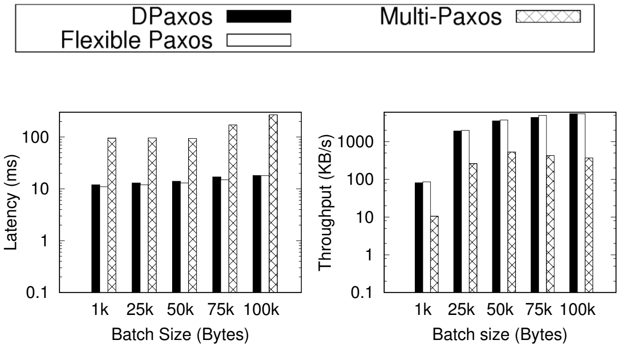


Figure 11: The effect of batching on performance

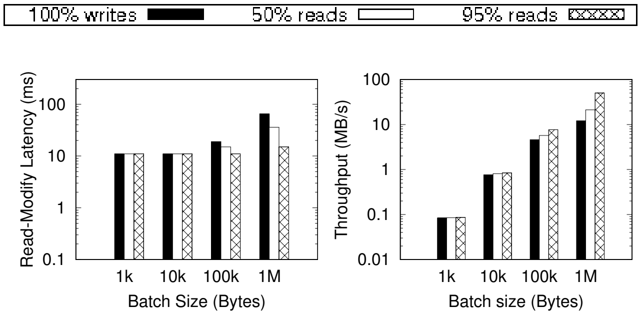


Figure 12: Scalability with local read-only requests

larger batches stresses the network I/O. Multi-Paxos thrashing behavior compared to DPaxos and Flexible Paxos is more severe because it communicates with a larger number of nodes in each round. A downside of batching is an increase of average latency. In the case of DPaxos and Flexible Paxos the latency increases from 11–12ms for 1 KB batches to 18ms for 100 KB batches. For Multi-Paxos, the latency increases from 95ms for 1 KB batches to 268ms for 100 KB batches. Interestingly, the latency of Multi-Paxos does not significantly change before the thrashing point (50 KB batches), maintaining a latency between 94–96ms.

To summarize, batching helps increase performance and utilize the communication links more efficiently. However, the trade-off is an increase in latency as batch sizes grow. Multi-Paxos suffer from thrashing behavior more rapidly than DPaxos and Flexible Paxos due to the larger communication requirements.

A.2 Read-only requests

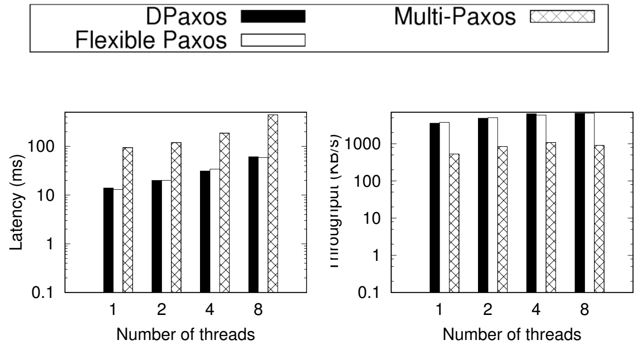
Read-only requests can be served from the leader without incurring the overhead of the Replication phase (Section 4.5). Throughout the other experimental evaluations, all transactions were read-modify transactions. Now, we introduce read-only transactions to the workload to measure the performance improvement gained from using master leases. Figure 12 compares three sets of workloads with DPaxos: a workload with 100% read-modify transactions (denoted in the figure 100% writes), a workload with 50% read-only transactions (denoted in the figure 50% reads), and a workload with 95% read-only transactions (denoted in the figure 95% reads). Also, we vary the batch size from 1 KB to 1 MB. In all these experiments, read-only transactions latency is less than 1ms. In comparison, the lowest latency of read-modify transactions is 11ms because of the time needed for the Replication phase. Other than the latency improvement, serving read-only transactions directly from the leader reduces the stress on compute and network resources. For example, in a batch of 1000 transactions, if half of the transactions are read-only, then the Replication phase only processes and replicates 500 transactions. Figure 12 presents this scaling behavior. With a small batch size (1 to 10 KB), there is no significant difference in the throughput of the different workloads. As we increase the batch sizes, we begin to notice the performance difference. With 100 KB batches, the workload with 50% read-only transactions achieves 24% higher throughput and the workload with 95% read-only transactions achieves 67% higher throughput. For the largest batch size we present (1 MB batches), the difference becomes larger. In that case, the workload with 50% read-only transactions achieves 75% higher throughput and the workload with 95% read-only transactions achieves 313% higher throughput. The reason for this performance difference is that the workloads with more read-modify transactions stress the compute and network resources more and begin to thrash earlier with larger batch sizes. An effect of this thrashing is shown as an increased in read-modify transactions latency. The workload with all read-modify transactions experiences a jump of latency from 11ms with 1 KB batches to 65ms with 1 MB batches, whereas the workload with 95% read-only requests maintains a read-modify transactions latency of 15ms with 1 MB batches.

A.3 Multi-programming level

We now explore another factor that affects throughput, which is the multi-programming level, defined as the number of slots that the proposer compete for simultaneously. For example, a multi-programming level of 4 means that the proposer can be deciding the values of 4 slots concurrently. We measure the effect of the multi-programming level in Figure 13, where we vary the level between 1 and 8. In this experiment, we only display the performance of the Virginia proposer. We chose Virginia because it the proposer that achieved the highest throughput for Multi-Paxos, and we wanted to compare the performance of DPaxos and Flexible Paxos with the best achieving proposer in Multi-Paxos. Also, we set the batch size to 50 KB, which maximizes Multi-Paxos performance without thrashing it. Increasing the multi-programming level from 1 to 8 improves throughput for DPaxos by 86%, for Flexible Paxos by 77%, and for Multi-Paxos by 71%. Like the batching experiments, Multi-Paxos experiences a thrashing behavior—this time the thrashing point is a multi-programming level of 4.

A.4 The effect of the Intents list

The Leader Election phase in DPaxos is subject to becoming more expensive in case there are old declared intents that have not been garbage collected, which lead to expanding Leader Election quorums unnecessarily. We quantify this effect in a set of experiments shown in Figure 14. In these experiments we do not garbage collect intents and intentionally place intents that cover zones, ranging from 1 zone to all 7 zones. Each point in the x-axis denotes how many zones are covered in the intents. For example, the case denoted 3 in the x-axis represent a case where there are three intents



(a) The latency of requests at each datacenter (b) The throughput in each datacenters

Figure 13: The effect of multi-programming on performance, where the number of threads is equivalent to the multi-programming level

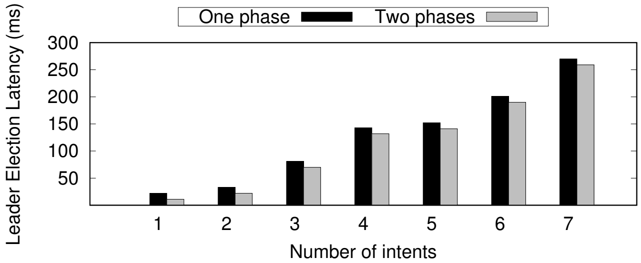


Figure 14: The Leader Election latency with a varying number of datacenters in the intent list

	C	O	V	T	I	S	M
C	0	19	62	113	134	183	249
O	19	0	117	104	133	161	221
V	62	117	0	172	81	244	182
T	113	104	172	0	214	67	124
I	134	133	81	214	0	179	120
S	183	161	244	67	179	0	58
M	249	221	182	124	120	58	0

Table 1: The average Round-Trip Times in milliseconds for every pair of the 7 datacenters (zones).

detected in three different zones. We order the places of these intents from the closer to the farther nodes. In these experiments, the proposer and the previous leader are both in California (but on different nodes). We show the latency of the original Expanding Quorum, where the intents are intersected in a second phase after Leader Election and the optimization to proactively intersect intents in the first round by sending redundant vote requests. The Leader Election latency increase depends on how far are the zones corresponding to the detected intents. The Leader Election latency ranges from 22ms to 270ms when two phases are used, and ranges between 11ms and 259ms when the two phases are combined. The benefit of the combination is to dilute the latency of the first phase in the latency of the second phase, in case the second phase has a higher latency.

B SURVEY OF RELATED WORK

B.1 Paxos Variants

In this section, we survey popular Paxos variants and Paxos-based systems.

(a) Multi-Datacenter Paxos Variants. There have been a number of Paxos variants and implementations that target multi-datacenter replication [5, 10, 18, 33, 41]. The main limitation of these solutions is that they rely on majority-based quorums for both Leader Election and Replication. This is true even for variants that specifically aim to reduce the inter-datacenter latency of Paxos such as EPaxos [33], and the adoption of Fast Paxos [24] by MDCC [18]. In these variants, majority or larger super majority quorums are used for replication to avoid the need for a centralized leader. Therefore, the replication phase becomes very expensive (spanning more than a majority of zones) compared to DPaxos. However, such approaches have the advantage over DPaxos that they bypass the leader election phase. Typically, the Replication phase occurs much more frequently than the Leader Election phase. In such cases, the benefit of small DPaxos Replication quorum outweighs the disadvantage of having to go through a Leader Election phase. (we compare DPaxos with leaderless Paxos variants in Section 5.3.)

(b) Flexible Paxos. Flexible Paxos [16] proposes the concept of non-majority quorums for Paxos (the inter-intersection condition in Definition 1.) This theoretical finding has enabled our adaptation of Flexible Paxos to implement Zone-centric quorums. However, this—as we describe in the paper—can only be done at the expense of Leader Election quorums that span all zones. What distinguishes DPaxos from Flexible Paxos is: (1) DPaxos is a practical deployment of flexible quorums for global-scale edge environment, (2) DPaxos proposes Expanding quorums that enable smaller Leader Election quorums than what Flexible Paxos can achieve, and (3) DPaxos proposes Leader Handoff to enable changing the location of the leader without a Leader Election phase. WPaxos [2] is a recently proposed adaptation of Flexible Paxos to geo-distributed systems. DPaxos is similar to WPaxos in that they utilize Flexible Paxos quorums to commit in nearby nodes rather than a majority. WPaxos proposes a novel *object stealing* technique with concurrent leaders at various locations. A leader A maintains ownership of data objects that are accessed close to A. Leader A can "steal"—via a Leader Election round—other leaders' objects if their access locality changed to be close to A. DPaxos can adopt this method to increase its adaptability to access locality. Likewise, WPaxos can also adopt DPaxos' Expanding Quorums and Leader Handoff approaches to overcome the expensive Leader Election inherited from Flexible Paxos.

(c) Reconfiguration and Hierarchical Paxos Variants. A possible approach to avoid inter-zone communication is to deploy the Paxos instance on $2f_d + 1$ nodes across $2f_z + 1$ zones. In such a case, like DPaxos, there will be no unnecessary inter-zone communication in the Replication phase. However, the main limitation of this approach is that only the current nodes in the zones are part of the Paxos instance. Thus, if nodes fail or users change location, a new leader for the deployment cannot be elected using a simple Leader Election round. Rather, a *reconfiguration* of the Paxos instance is needed to change the set of participating nodes to the new ones in the new location or zone. There are some Paxos variants that deal specifically with this issue of managing reconfiguration, such as Vertical Paxos [25], Stoppable Paxos [26], and Cheap Paxos [27].

The common factor behind these solutions is that they separate the control from the operation in Paxos, where there is an *auxiliary Paxos instance* that manages the different configurations for all partitions and a Paxos instance for each partition.

Any configuration change, to replace nodes or change the set of participants, would be done through the auxiliary Paxos instance. The auxiliary Paxos instance can either be centralized in a single zone or distributed across zones. If the auxiliary Paxos instance is centralized, then any configuration change at another zone would experience inter-zone latency. Otherwise, if it is distributed across zones, then all configuration updates would incur inter-zone communication. This makes such configuration changes more expensive than a DPaxos Leader Election round in terms of communication overhead and latency. Likewise, Paxos can be used as a control layer to manage configuration changes. ZooKeeper [17], for example, is typically used in this way. It is possible to consider a ZooKeeper (or another Paxos-based system) instance being used to maintain the configuration of the current Replication quorum. This makes it similar to Leader Zone Quorums in that it establishes a centralized point to manage the location of the Replication quorum. However, like other hierarchical approaches, the fixed location of the nodes in the control Paxos instance leads to high latency with moving workload as the control Paxos instance cannot (or incurs high overhead) to relocate. This also applies to primary/backup systems that use a designated node for reconfiguration [47].

(d) Other Paxos variants. Generalized Paxos [23] allows values that are not conflicting with each other to be decided concurrently, thus improving throughput. Generalized Paxos is an orthogonal technique to DPaxos that can be applied to it to improve concurrency. Paxos-CP [41] proposes optimizations to increase the concurrency of Multi-Paxos for transaction processing by combining different transactions in the same slot if they do not conflict. Compared to DPaxos, Paxos-CP requires a round to a majority to decide a value. Some other Paxos variants tackle the problem of load balancing such as Mencius [32, 48] and S-Paxos [6]. Network-Ordered Paxos (NOPaxos) [28] proposes leveraging network-level ordering to design a simpler more efficient Paxos variant.

B.2 Data Management for edge computing

There has been previous work on designing data management solutions for edge computing [11, 29, 43]. The edge computing data management model introduces new challenges and problems and these studies tackle subsets of them. For example, EdgeX [43] manages the process of offloading partitions to edge datacenters in a consistent manner. Also, Eyal et. al. [11] propose mechanisms to improve the consistency of caches in the edge. Lin et. al. [29] propose coordination techniques for data management on the edge where replicas guarantee snapshot isolation.

B.3 Multi-Level Quorum Consensus

Voting and quorums are ideas that were introduced in the late 70s in the context of data management [12]. Since then, a plethora of work studied quorum allocation to optimize quorum sizes. Examples include grid quorums [30], tree quorums [1], and weighted quorums [12]. However, the closest quorum formulation to our work is hierarchical quorums [19, 20]. In this method, nodes are arranged in a virtual hierarchy. Then, a quorum is represented in terms of "groups" of nodes rather than the nodes themselves, *e.g.*, a vote from a majority of groups rather than a majority of nodes. Because a group of nodes may delegate their votes to a subset of the group, it is possible to form quorums with smaller sizes than a majority.

B.4 Global-Scale Data Management

Optimizing the performance—especially reducing latency—of strongly consistent geo-distributed transactions has been the focus of many pieces of work [3, 4, 10, 13, 18, 31, 35–42, 45, 51]. Many of the proposed protocols rely on Paxos as a building block for synchronous wide-area replication [3, 10, 13, 18, 31, 40, 41, 45, 46]. For these protocols, DPaxos offers a more efficient alternative to Paxos especially in emerging edge data management environments.

C GARBAGE COLLECTION CORRECTNESS

In this section, we present the proof of Theorem 3:

Proof. Assume to the contrary that the replication quorum of an intent with proposal id p that is less than $\mathcal{P}$ has accepted a proposal with proposal id p.

The first outcome of this assumption is that every node in the intent's replication quorum has responded with an accept message to a propose message with proposal id p. This also means that there is a node, $n(p)$, that successfully performed a Leader Election round with proposal id p and received promise messages from a Leader Election quorum.

Also, the value $\mathcal{P}$ indicates that, by definition, there has been a node, $n(\mathcal{P})$, that sent propose messages. Because it has sent propose messages, node $n(\mathcal{P})$ must have successfully performed a Leader Election round, where it received promise messages from a Leader Election quorum.

So far, we know that $n(p)$ and $n(\mathcal{P})$ have each acquired the votes of a Leader Election quorum. By definition, any two Leader Election quorums intersect. This means that there is a node $\mathcal{L}$ that is in both $n(p)$'s and $n(\mathcal{P})$'s Leader Election quorums. Therefore, $\mathcal{L}$ has responded with a promise message to both nodes. Because $p < \mathcal{P}$, $\mathcal{L}$ must have responded to $n(p)$'s prepare message first. Now, consider when node $\mathcal{L}$ receives $n(\mathcal{P})$'s prepare message. There are two cases:

- Case 1: $n(p)$'s intent is still in $\mathcal{L}$ and is piggybacked in the response (the promise message) back to $n(\mathcal{P})$. When $n(\mathcal{P})$ receives the intent, it expands the Leader Election quorum to intersect with $n(p)$ intent's replication quorum. We know that $n(\mathcal{P})$ successfully gets a promise message from at least one node $\mathcal{I}$ in $n(p)$'s replication quorum (because it successfully completed the Leader Election phase and sent propose messages). We also know that it must have received the promise message before it moved to the Replication phase and sent propose messages. Therefore, node $\mathcal{I}$ would not accept any new messages from $n(p)$ with proposal id p, which is a contradiction to our assumption.
- Case 2: $n(p)$'s intent in $\mathcal{L}$ has already been garbage collected. This means that there is another node, $n(\mathcal{P}2)$, that has encountered case 1 above and determined that the intent's replication quorum cannot accept $n(p)$'s proposals.

We arrive at a contradiction in all cases, which proves the theorem. □