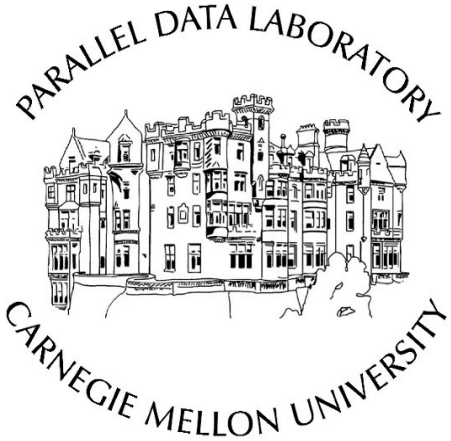

Paxos Quorum Leases: Fast Reads Without Sacrificing Writes

Iulian Moraru[1], David G. Andersen[1], Michael Kaminsky[2]

[1] Carnegie Mellon University, [2] Intel Labs

CMU-PDL-14-105

May 2014

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

This paper describes quorum leases*, a new technique that allows Paxos-based systems to perform reads with high throughput and low latency. Quorum leases do not sacrifice consistency or write latency, and have only minimal impact on system availability. Quorum leases allow a majority of replicas to perform strongly consistent local reads, which substantially reduces read latency at those replicas (e.g., by two orders of magnitude in wide-area scenarios). Previous techniques for performing local reads in Paxos systems either (a) sacrifice consistency; (b) allow only one replica to read locally; or (c) decrease the availability of the system and increase the latency of updates by requiring all replicas to be notified synchronously. We describe the design of quorum leases and evaluate their benefits compared to previous approaches through an implementation running in five geo-distributed Amazon EC2 datacenters.*

Acknowledgements: This work was supported by Intel Science & Technology Center for Cloud Computing, Google, and the National Science Foundation under award CCF-0964474. We would like to thank the members and companies of the PDL Consortium (including Actifio, American Power Conversion, EMC Corporation, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc., Oracle Corporation, Panasas, Riverbed, Samsung Information Systems America, Seagate Technology, STEC, Inc., Symantec Corporation, VMware, Inc., and Western Digital) for their interest, insights, feedback, and support.

Keywords: Distributed Consensus, Paxos

1 Introduction

Paxos [18] and its numerous variants for state machine replication have become increasingly important for large-scale Internet services. They extend to span massive datacenters, geo-replicate within and across continents, and handle tens of billions or operations each day [8]. Many Paxos-based systems improve performance by using some form of read leases, in which one or several replicas can satisfy read requests locally without having to commit an operation using the relatively more expensive Paxos protocol. This optimization obviously improves read throughput and latency, but also improves write performance by reducing the total number of operations that must be handled by the Paxos replicas.

A variety of approaches to read leases have been used in practical systems. Most Paxos-based systems use the Multi-Paxos optimization, in which one node is temporarily selected as the "stable leader" and is responsible for orchestrating all operations. Absent failures, Multi-Paxos enables operations to commit in a single round trip from the leader to a majority of replicas. In a Multi-Paxos environment, the most common read lease optimization is to grant the stable leader a read lease on all objects comprising the state—files, key-value pairs or relational database records, depending on the application. As a result, reads can be handled with a single round-trip to the leader, and writes incur no additional slowdown, compared to a Paxos system that does not use leases.

Other systems, such as Google's Megastore [3], instead grant a read lease to *all* nodes. This decision aggressively optimizes for reads at the expense of write latency: all reads can be handled locally at a replica with no inter-replica traffic whatsoever, but writes now involve at least one round-trip to every replica (instead of just the nearest majority).

In this paper, we argue that there is an overlooked alternative that is a more natural fit to the structure of Paxos: *quorum leases*. In this model, a lease for each object in the system could be granted to different subsets of nodes. The size and composition of these subsets is selected either based upon how frequently each replica reads the objects in question (for best read performance) or based upon their proximity to the leader (to improve read performance without slowing write performance). A particularly suitable size for this subset is that of a Paxos quorum: we claim that quorum leases are a "natural" fit to Paxos because writes in Paxos must, by definition, synchronously contact at least a simple majority of nodes anyway. Thus, the lease revocation and the Paxos messages can be combined, often resulting in no additional overhead or delays for handling a leased object.

Despite the intuitiveness of this approach, implementing quorum leases is nontrivial. Compared to approaches in which the set of nodes with the lease is fixed—either a single master or all replicas—an implementation of quorum leases must be able to consistently determine which objects belong to which lease quorum, automatically determine appropriate lease durations, and efficiently refresh the leases in a way that balances overhead, a high hit rate on leased objects, and rapid lease expiration in the event of a node or network failure. Solving these problems and evaluating the resulting benefits is the primary objective of this work.

Our results (Section 5) suggest that quorum leases present an extremely attractive middle ground for practical paxos-based systems: running the YCSB benchmark workload on a geo-distributed deployment of five-replica Multi-Paxos with quorum leases, over 80% of reads are handled locally, with no wide-area traffic, at each replica, and, and over 70% of writes have the smallest client-observed latency attainable in a Multi-Paxos system, for the given geographic configuration. In contrast, a single leader lease system achieves 100% of minimal latency writes, but only 20% of reads are local (those performed by the stable Multi-Paxos leader).

2 Overview

We begin with a short overview of the Paxos protocol, and then we present the intuition of applying quorum leases to Paxos.

2.1 Paxos Overview

The Paxos protocol [19] is used to implement state machine replication by making a set of possibly faulty distributed replicas execute the same commands in the same order. Because each replica behaves like a state machine, all non-faulty replicas will transition through the same sequence of states as a result. To be able to make progress, fewer than half the replicas can be faulty—if N is the total number of replicas, at least $\lfloor N/2 \rfloor + 1$ must be working correctly. Paxos handles only non-Byzantine failures: replicas may crash or fail to respond indefinitely, but they cannot respond in ways that do not conform to the protocol.

The original and most general specification of Paxos uses processors with different roles: leaders, acceptors and learners. In practice, these roles are collapsed, and a replica plays multiple roles at the same time. Because this corresponds to practical deployments, we present our design in terms of replicas with identical capabilities.

The execution of a replicated state machine that uses Paxos proceeds as a series of independent *instances*, where the outcome[1] of each Paxos instance is the agreement on which command must be executed by every replica at a particular position in the linear sequence of commands. The voting process for one instance may happen concurrently with voting processes for other instances, but does not interfere with them.

Upon receiving a command request from a client, a replica first tries to become the *leader* of an instance corresponding to a command position it believes has not been used. It does so by sending *Prepare* messages to at least a majority of replicas (possibly including itself). A reply to a *Prepare* contains the command that the replying replica believes may have already been chosen in this instance (in which case the new leader must use that command instead of the newly proposed one), and also constitutes a promise not to acknowledge older messages from previous leaders. If the new leader receives at least $\lfloor N/2 \rfloor + 1$ acknowledgements for *Prepare* messages, it will proceed to propose its command by sending *Accept* messages to a majority of peers; if these messages are also acknowledged by a majority, the leader commits the command locally, and then asynchronously notifies all its peers and the client.

Because this canonical mode of operation requires at least two rounds of communication (two round trips) to commit a command—and possibly more in the case of dueling leaders—the widely used "Multi-Paxos" optimization [19] designates a *stable leader* replica (i.e., the *distinguished proposer)*. A replica becomes the stable leader by running the prepare phase for a large (possibly infinite) number of instances at the same time, taking ownership of all of them. In steady state, clients send commands only to the stable leader, which directly proposes them in the instances it already owns (without running the prepare phase first). When a non-leader replica suspects that the leader has failed, it tries to become the new leader by taking ownership of all the instances it believes have not yet been finalized.

Figure 1 presents a time diagram depicting the message exchanges in a Multi-Paxos system, in steady state.

2.2 Quorum Leases: Intuition

Recall that a lease is a time-limited promise from one process to another to not modify an object during the *lease duration*. Leases are often coupled with a *revocation* mechanism: If the lease grantor wishes to

[1] By FLP [10], it is impossible to guarantee termination, although under realistic conditions, using timeouts and retransmitting messages ensures termination with high probability.

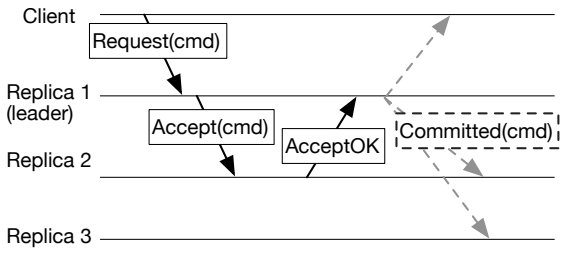

Figure 1: Steady state interaction in Multi-Paxos. The asynchronous messages are represented as dashed arrows.

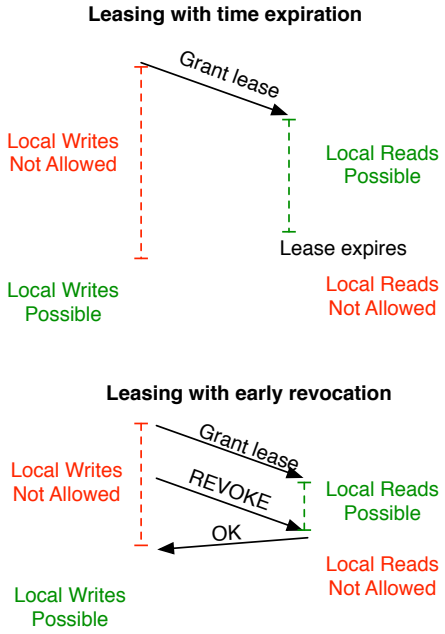

Figure 2: Leasing with and without revocation.

modify the object before the lease has expired, it must contact each lease holder and receive confirmation that the holder will stop using its local copy of the object, as shown in Figure 2.

Quorum leases intertwine the idea of leasing with the natural set of nodes that must be contacted for a Paxos write, bundling the Paxos write operation with the lease revocation. In Paxos, a replica can commit a command only after a majority quorum—possibly including itself—has acknowledged the command. As is apparent from the revocation example in Figure 2, if the write quorum includes *all* nodes that hold leases on the object to be modified, then receiving acknowledgements from the write quorum *also* means that the lease has been properly revoked at all replicas.

Of course, this simple design has several complications: (1) if any member of the quorum becomes unavailable, no command can be committed until the lease expires; and (2) replicas outside the quorum cannot perform local reads.

We solve the first problem by carefully choosing the lease duration relative to the round-trip time between the replicas. A quorum is notified synchronously only for a set amount of time, after which none of its members can rely on their state being updated synchronously anymore. If a quorum member crashes, the system will be unavailable only until the lease expires.

To reduce the opportunity cost of not leasing to the full set of nodes, we grant leases on a per-object basis, where each lease might have a different set of nodes holding it. In this way, while not *all* nodes can read locally, the nodes generating the most read traffic for each object can do so. This design is particularly appropriate when the popularity of each object differs substantially across replicas—such as might be the case, e.g., in the popularity of users across a geo-distributed social network.

In conclusion, quorum leases take advantage of the existing communication patterns in Paxos to allow replicas to perform local reads, without substantially decreasing the availability of the system, and without significantly increasing write latency.

3 Design

We begin by describing the assumptions about the Paxos systems that use quorum leases. We then describe our quorum leases design goals and motivate our design choices.

3.1 Assumptions

Communication between the nodes is asynchronous: messages may be lost or delayed indefinitely. Replicas *do not synchronize clocks*, but their clock rates are assumed to be similar, such that a modest guard time can account for clock drift over a short interval. Failures are non-Byzantine: replicas can crash or fail to respond indefinitely, but they will not take actions that do not conform to the protocol.

We assume that the replicated state consists of multiple objects that can be updated and read separately. Clients submit operations that specify which objects will be updated, and queries for reading object values. Multiple updates and queries can be batched in the same command.

3.2 Design Goals

A *quorum lease* provides a subset of Paxos replicas the guarantee that they will be notified synchronously of any update to a particular set of objects before those updates are committed by any replica. Thus, let $\mathcal{R}$ be the set of all replicas in a Paxos group, and let $\mathcal{O}$ be the set of all objects replicated by this Paxos group. A quorum lease is a pair (Q, O), where $Q \subseteq \mathcal{R}$ and $O \subseteq \mathcal{O}$. We call Q the *lease quorum* for this lease, and O the set of *granted objects*. Every replica $r \in Q$ is a *lease holder*.

Unlike a Paxos quorum, which must include a majority of replicas, a lease quorum can be of any size—e.g., it can contain fewer than half the replicas.

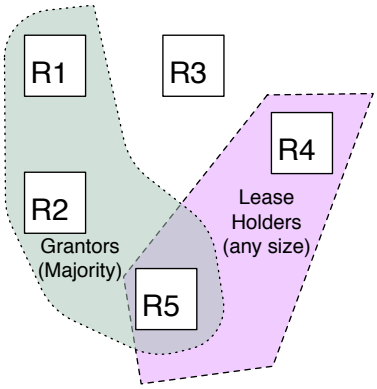

Figure 3: An example lease in which a majority of replicas (R1, R2, and R5) have granted leases to two lease holders (R4 and R5).

A lease becomes active after a majority of replicas (i.e., at least $\lfloor N/2 \rfloor + 1$ replicas, where $N = |\mathcal{R}|$) have *granted* the lease to at least one lease holder. An example of such a lease is shown in Figure 3, where a majority have granted read leases to two nodes in the set. A replica g grants a lease (Q, O) to a replica $r \in Q$ by making a promise:

1. to notify r synchronously before committing any update to any object in O that g proposes (i.e., g must not commit until it receives a message from r in response to its notification), AND

2. to acknowledge Accept and Prepare messages[2] for updates to objects in O only with the condition that the proposer must also notify r synchronously before committing these updates.

To enforce the first condition without extra communication, g can include r in any Paxos quorum of acceptors before committing a Paxos command that it proposes (i.e., for which it is the leader). To enforce the second condition, every replica must attach enough information to its replies to the Paxos Accept and Prepare messages from a proposer that the proposer can determine which replicas the responder has granted leases to; the proposer uses this information to synchronously notify the lease holders before committing. We explain in Section 3.4 how to implement this exchange efficiently by adding only a short lease identifier to each message.

To prevent unbounded periods of unavailability, a promise is valid for only a set amount of time, after which it expires. In a correct implementation, a promise must expire at the lease holder before expiring at the grantor. When a lease holder has fewer than $\lfloor N/2 \rfloor$ valid promises from different replicas (the holder itself counts as an implicit grantor), the lease is said to be inactive for that holder.

This mechanism achieves the following: while the lease is active at a lease holder, that lease holder can assess whether an update to any of the leased objects is ongoing (i.e., in the process of being committed), and if not, the lease holder can read the most up-to-date value of that object from its local store.

A quorum lease can be granted to any subset of replicas, of any size. However, because updates in Paxos must be accepted synchronously by a majority of replicas even when not using leases, it is advantageous for both latency and availability to make lease quorums be simple majorities ($\lfloor N/2 \rfloor + 1$ out of the total of N replicas). For reduced write latency, it is also useful that every lease quorum include the current distinguished proposer (i.e., the current stable leader, if the Paxos variant used relies on a stable leader—such as, for example, Multi-Paxos).

[2]The condition for acknowledging Prepare messages applies only to instances where the grantor has already accepted an update for an object in O.

Different Paxos replicas may need to read different sets of objects at different times. We therefore wish to be able to update both the set of leased objects, and the lease quorums. We call the totality of quorum leases agreed upon at any given moment a *lease configuration*, and we use Paxos to achieve consensus on lease configuration changes. Conceptually, there is a separate, independent Paxos-replicated state machine for the configuration state; in our implementation, the configuration Paxos instantiation runs on the same nodes as the main Paxos instantiation. Because replica clocks are not synchronized, we must establish timing dependencies when granting and refreshing leases. We do this through separate communication, independent of the lease configuration change protocol (see below).

Because an unresponsive lease holder will prevent updates to the leased objects until the lease expires, it is useful for leases to be granted for short periods of time, which are refreshed before they expire.

3.3 Design Overview

Although we believe that quorum leases apply to any variant of Paxos, we focus for clarity on the most popular: Multi-Paxos.

While using quorum leases, the replicas of a Paxos system communicate using two categories of messages: (1) The normal Paxos messages for choosing commands; and (2) Lease management messages. We separate these in both design and implementation to make it easier to reuse the leases with other Paxos variants, and to modify the lease management protocol.

Lease management consists of two message sub-types: (2a) Paxos messages for agreeing on lease configuration changes—what are the quorums and what is the set of object IDs granted to each quorum; and (2b) messages for granting and refreshing leases.

To avoid a dependency on external clock synchronization, we set lease timers based upon the causal ordering of messaging events using a lightweight peer-to-peer protocol described in detail in Section 3.5. This protocol efficiently combines the computation of lease timers and the granting and refreshing of leases.

Separating lease configuration and granting conveys several advantages. First, it means that the granting protocol could be improved independent of the rest of the system to take advantage of, e.g., hardware clock synchronization or stronger assumptions about delays based upon knowledge of the physical connections between machines. Importantly, the simplicity of the lease granting protocol also makes short lease intervals feasible, which has important benefits for availability. The lease renewal messages are short and can be piggybacked on existing traffic, as they refer to the current lease configuration through a short configuration ID, instead of needing to explicitly describe quorums and granted objects as the lease configuration messages must.

3.4 Lease Configurations

A lease configuration describes the quorum composition and the set of granted objects for all the quorum leases in a Paxos system at a given time.

A lease configuration is built incrementally from a sequence of lease configuration changes. Replicas agree on lease configuration changes through Paxos—they participate in Paxos instances that are separate from the normal command-choosing instances. The result is a sequence of lease configuration changes that all (non-faulty) replicas agree upon. A lease configuration is then simply referred to by the instance number of the latest change.

Our basic implementation of quorum leases strives to assign leases such that: (a) the lease is granted to a simple majority of nodes; and (b) the number of locally-satisfied read operations is maximized; and (c) the total number of leases being managed is modest. [3]

[3] Many other optimization goals are both possible and reasonable; exploring them more deeply is an interesting avenue of future work.

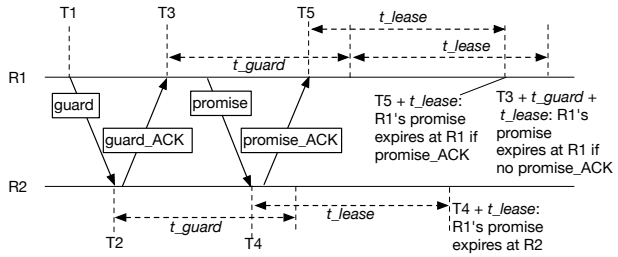

Figure 4: When clocks are not synchronized, the lease activating procedure uses acknowledgements to safely manage lease intervals. In this diagram, the lease duration is *t_lease*. A grantor (R1) will only send a promise if its guard is acknowledged by the holder (R2). The grantor can thus bound the promise duration even if the holder does not reply to the promise.

It accomplishes this by having replicas track the frequency of reads. The current Multi-Paxos leader replica periodically gathers this information and determines the next lease configuration such that each object is granted to the majority quorum that consists of the stable leader and the $\lfloor N/2 \rfloor$ additional replicas that have read the object most often; it then proposes this new configuration—a set of (quorum, object ID list) pairs—in one of the special Paxos instances. Because it is already guaranteed to see all writes by virtue of being the Multi-Paxos leader, the stable leader is granted a *default lease* that covers every object not in another lease, and is also included in every lease quorum.

There are many possible ways to maintain read statistics. In our current implementation, when a replica receives a read request that it cannot service locally, it forwards it to the stable Multi-Paxos leader. The leader counts the number of forwarded read requests for each object-replica pair. If the number of reads from a given replica is larger than the number of reads from another replica previously included as a lease holder for the object, the leader will include the new replica as a lease holder in the next lease configuration update.

Lease configuration changes happen sufficiently infrequently that they can be written to a stable log without affecting the performance of the system.

3.5 Activating Leases

Replicas agree on a lease configuration as described in the previous section, but the quorum leases that constitute this configuration become active only after being granted as explained in this section.

The lease configuration covers all leases granted on all objects (which may involve many different sets of holder replicas), and so replicas affirm this configuration in an all-to-all manner. Every replica sends a *promise* to every other replica in the system (thus becoming a grantor). The promise includes the number of the most recent lease configuration that the grantor is aware of, the lease duration, and a timestamp. A receiver (i.e., a lease holder) rejects promises for lease configurations older than the newest one it is aware of.

A lease represents a time during which the grantor will not modify an object without contacting the holder, which gives the holder permission to read that object locally. For safety, the grantor's "will not modify" window of time must be inclusive of the holder's "can read locally" window of time. For high availability, these times should be as short as possible, so that a failed holder can only block writes for a short period of time. These goals are accomplished by the lease establishing and renewal protocol described in Figures 4 and 5.

Before sending a promise, the grantor sends the holder a *guard* message, which the holder must acknowledge. The guard specifies a time duration t_{guard}. The subsequent promise contains a lease duration t_{lease}. Importantly, the promise is *only* valid if received by the holder before t_{guard} has expired. This ensures

that even if the holder does not respond to the promise, the grantor knows that the holder will not believe it has a lease subsequent to $t_{\text{guard}} + t_{\text{lease}}$ seconds after it received the guard ACK.

A holder starts a lease timer as soon as it receives a promise. The promise expires after its specified lease duration (t_{lease}) has passed. A lease holder can consider the lease active while there are at least $\lfloor N/2 \rfloor$ promises received from different peers that have not yet expired.

When renewing active leases, there is no need to send the guard anymore—the holder can consider a renewal (i.e., a new promise extending the lease) valid only if it receives it before the previous promise from the same grantor has expired. This lets the grantor relinquish its new promise after a time t_{lease} from the expiration of the previous promise, even if the holder does not reply.

When a grantor becomes aware that a new lease configuration has been agreed upon while its most recent promises are still valid, it can take one of two approaches: (1) the grantor can let its current promises expire, and then send promises for the new configuration; or (2) the grantor can immediately send promises for the current configuration, but while both sets of promises are valid, it must abide by the lease rules of both the previous and the current configuration.[4] For simplicity, our current implementation takes the first approach.

The lease establishing and renewing logic is described in pseudocode in Figure 5.

3.6 Ensuring Strong Consistency

Paxos and Multi-Paxos provide strong consistency: operations are strictly serializable (they are both serializable and the temporal order of non-overlapping operations is respected). In this section we show that quorum leases maintain this consistency guarantee.

Write-only and composite read-write operations will be committed through the normal Paxos protocol and executed atomically. They will thus be strictly serializable. Every simple or composite read-only operation will either be serviced atomically by a replica that holds leases for all the objects in question, or will be committed through normal Paxos if no such replica exists. Thus, the system ensures serializability.

To ensure *strict* serializability, notice that it is necessary and sufficient to show that given a read-write or write-only operation W and a read-only operation R, where the write set of W intersects with the read set of R, then the system observes their temporal order. That is to say (1) if R completes before W begins at any replica, R does not observe W; and (2) if W is committed at any replica before R is received by any replica, R observes W. If R is committed through Paxos, this is true by virtue of the Paxos protocol guarantees. We must show that the property also holds when R is serviced locally by some replica.

If R completes before W begins, the property holds trivially: R cannot return a version that does not yet exist anywhere. We are therefore left with the case where W was committed before R was received.

The replica that services R locally (we will call it reader_R) can only do so if it has a lease for the objects that R refers to. We must therefore be in one of the following situations:

Case 1: reader_R acquired the lease before W began. In this case, reader_R's lease was active throughout W's Paxos commit process. Because the lease is active, by the causal ordering of grantors and holders starting their lease timers, it must be the case that at least $\lfloor N/2 \rfloor + 1$ replicas (possibly including reader_R) have granted reader_R the lease, and their promises are still active (i.e., binding). The quorum of replicas that accepted W will intersect this quorum of grantors in at least one replica grantor_R. Because grantor_R accepts W, it must be the case that W's proposer will learn that there is an active lease when grantor_R replies to its Accept. Therefore, W's proposer will know that it must notify reader_R before committing W (even if this proposer has not made any promises itself), and reader_R must execute W before executing any further read, including R. In conclusion, R will observe W.

[4] For example, if an object has been granted to a different quorum, the grantor must ensure that it notifies synchronously every replica in the union of the two quorums before committing an update to the object.

Establishing leases

Every replica R becomes a grantor:

1: send *Guard(guard_duration)* to every other replica
2: **for** every *GuardACK* from any replica H **do**
3: set $grant\_timer_R[H]$ to $guard\_duration + lease\_duration$
4: send *Promise(lease_duration)* to H
5: **for** every *PromiseReply* from any replica H **do**
6: **if** reply received before $grant\_timer_R[H]$ expired **then**
7: set $grant\_timer_R[H]$ to *lease_duration*

Any replica H, on receiving a $Guard(guard\_duration)$ from a replica R:

8: set $guard\_timer_H[R]$ to *guard_duration*
9: reply with a *GuardACK*
10: wait for a *Promise(lease_duration)* from R
11: **if** *Promise* received before $guard\_timer_H[R]$ expires **then**
12: set $lease\_timer_H[R]$ to *lease_duration*
13: reply with *PromiseReply* to R

Renewing leases

Every replica R that is a grantor:

14: **for** every replica H for which $grant\_timer_R[H]$ has not expired **do**
15: extend $grant\_timer_R[H]$ by $lease\_duration$
16: send *Promise(lease_duration)* to H
17: **for** every *PromiseReply* from any replica H **do**
18: **if** reply received before $grant\_timer_R[H]$ expired **then**
19: set $grant\_timer_R[H]$ to *lease_duration*

Any replica H, on receiving a $Promise(lease\_duration)$ from a replica R:

20: **if** *Promise* received before $lease\_timer_H[R]$ expires **then**
21: set $lease\_timer_H[R]$ to *lease_duration*
22: reply with *PromiseReply* to R

{A lease holder H can consider the lease active if at least $\lfloor N/2 \rfloor$ promises from different replicas have yet to expire (where N is the total number of replicas).}

Figure 5: Establishing and renewing quorum leases.

Case 2: $\mathbf{reader}_R$ acquired the lease after W began. This case has three sub-cases: (1) If there exists an acceptor that is part of W's Paxos quorum and that grants reader_R the lease before accepting W, then this scenario reduces to the previous case. (2) If reader_R itself accepts W before its lease becomes active, it must execute W before R. (3) There exists an acceptor $\mathrm{grantor}_R$ that first accepts W and then makes a promise that reader_R takes into account when activating its lease. Because promises contain the index of the most recent accepted Paxos instance at the grantor (therefore at least as recent as W's instance at $\mathrm{grantor}_R$), reader_R knows that it must wait for commits for all instances up to and including W's instance before replying to any read (including R). These three sub-cases are exhaustive.

In conclusion, a Paxos system that implements quorum read leases ensures strict serializability.

3.7 Recovering after a Replica Failure

The failure of a lease holder will prevent the system from committing updates to any object for which the lease holder has a lease until enough of the promises made to it expire. The system can resume committing once a majority of replicas are no longer bound to notify the faulty replica synchronously. In practice, the time for which it is blocked is on the order of a few to ten seconds, depending on the round-trip time between

the replicas, as we analyze more carefully in Section 3.8.

A grantor suspects that a replica may have failed if that replica stops replying to its promises or heartbeat messages (common in many Paxos implementations). After a grace period, the grantor will stop trying to renew its promises, and it will let the ones made so far expire. In the meantime, the grantor will request a special lease configuration update that specifies that the replica suspected of failure should be excluded from all quorums it was part of. Replicas that switch to this new configuration no longer need to synchronously notify the possibly-failed replica of updates, and the system can safely resume using leases.

A replica that rejoins the replica set after a failure must wait for t_{wait} (Section 3.8) seconds before it can accept and/or propose any commands to ensure that all of its promises have expired.

3.8 Lease Time and Failures Analysis

In this section we analyze the relationship between the inter-replica RTTs, the lease duration, and the maximum window of write unavailability after a lease holder crashes.

The guard period t_{guard} (Section 3.5) must be larger than the maximum round-trip time between any two replicas; otherwise, the soon-to-be lease holder will reject the subsequent promise when it arrives. The lease duration t_{lease}, on the other hand, can be arbitrarily small because grantors can send the lease renewal traffic "blind"—that is, without knowing whether or not its previous lease renewals had been received successfully. However, for renewals to be consistently successful even when messages are lost and retransmitted, the lease duration should be higher than one RTT.

A grantor will stop issuing new leases once it believes a replica is down, as indicated by a failure to reply to some previous message. Such a message loss can only be detected after (at least) one round-trip time between the grantor and grantee. In practice, a grantor will likely wait some additional time before believing a replica failed. We term this total time, from the instant that a replica could have failed to the time its grantor stops issuing lease renewals, the "grace period."

At the end of the grace period, the grantors will conclude that the node in question has crashed. The grantors can exclude the crashed node from the lease configuration state immediately, but they cannot update any objects leased by the crashed node until they can be certain the crashed (or partitioned) node will not read its objects locally. We term this time t_{wait}, and it is calculated as follows:

Let f_{renew} be the frequency with which each grantor sends renewals; then, each grantor may send up to $t_{\text{grace}} \times f_{\text{renew}}$ unacknowledged renewals. In the worst case, the holder may have received each renewal exactly before the lease timer expired. Therefore, the grantor must assume that the lease could have extended to $t_{\text{grace}} \times f_{\text{renew}} \times t_{\text{lease}}$ seconds. Finally, the time that the grantors (and therefore the system, because grantors form a majority along with the crashed replica) will be unable to update the objects leased by the crashed replica will be $t_{\text{wait}} = t_{\text{lease}} \times (t_{\text{grace}} \times f_{\text{renew}} + 1)$. After t_{grace} expires, but before t_{wait} expires, the live replicas will initiate a lease configuration change (Section 3.7) so that they can resume using leases after t_{wait} expires.

If the Paxos leader fails, the unavailability time will be the maximum of the time to elect a leader and t_{wait}.

3.9 Multi-object Operations and Batching

With quorum leases, different objects may be granted to different quorums. Therefore, multi-object update operations must be synchronously acknowledged by a super-quorum—the union of all quorums that lease an object updated by the multi-object operation. If such operations are frequent, this may affect the performance and the availability of the system. A possible solution is to track those objects that are frequently updated together and ensure they are always granted to the same quorum.

A similar problem arises with batching. Batching increases the throughput of Paxos systems by grouping multiple concurrent operations in a single Paxos command. If these operations update objects leased by different quorums, the batch must be accepted synchronously by the union of all corresponding quorums. To avoid this, replicas must separate objects granted to different quorums into different batches before proposing them. Under heavy request load (usually the situation that warrants batching to sustain a high throughput) there will usually be enough operations of each type for batching to still be effective.

4 Implementation

We implemented quorum leases, classic leader leases and Megastore-type leases within an existing Multi-Paxos system written in Go. Because our main focus is on the message and round-trip time reductions (or increases) due to leasing in the wide area, we usually run the system at throughput levels where the implementation details are not the bottleneck. This reduces the importance of a specific choice of Paxos framework.

Because a major focus for all leasing strategies is to reduce latency in the wide area, we implemented as part of our baseline a latency optimization described by Castro [5]. This optimization reduces the commit latency perceived by clients that are co-located with a replica other than the Multi-Paxos stable leader, by having other replicas transmit their AcceptReplies to both the stable leader and to the replica near the client. Thus, in the common case, the client does not need to wait the additional time for a message to come back from the stable leader, which reduces commit time from four one-way delays to three.

We implemented this optimization because, while it does not appear to be commonly deployed, it is a straightforward algorithmic tweak that reduces commit latency and therefore more accurately represents the state of the art of the write latency that can be achieved by a Paxos-based system. This optimization benefits all three implementations (leader leases, Megastore-type leases, and quorum leases), but in some cases confers slightly more of an advantage to traditional leader leases than to quorum or Megastore-leases.[5]

5 Evaluation

We run our implementations of quorum leases, classic leader leases and Megastore-type leases, both in a single Amazon EC2 cluster, and in a geo-distributed setting: five Multi-Paxos replicas run in five Amazon EC2 datacenters, located in Virginia, Northern California, Oregon, Ireland and Japan. Ten clients are co-located (i.e., in the same datacenter) with each replica. Replicas and clients run on large Amazon EC2 instances: two 64-bit virtual cores with two EC2 Compute Units each and 7.5 GB of memory. The typical RTT in an EC2 cluster is 0.4 ms. The round-trip times between datacenters are summarized in Table 1.

5.1 The Workload

In our experiments, we use Multi-Paxos to replicate a key-value store. Lacking access to, e.g., user traces from a major Internet service, we use the YCSB [7] key-value workload to benchmark it.[6] Every client in our system proposes Put and Get operations with keys drawn from either a Zipf distribution (with an exponent of 0.99—the default YCSB implementation of a Zipf generator) or a uniform distribution. For the Zipf

[5] Without leases, the near-client replica will be able to commit as soon as it receives $\lfloor N/2 \rfloor$ AcceptReplies, or $\lfloor N/2 \rfloor - 1$ AcceptReplies and an Accept from the leader (because it is implicit that the leader must have accepted too). With leases, the commit condition is more strict: the replica closest to the client can commit an operation after receiving Accept or AcceptReplies messages from all the replicas that hold the lease for the objects updated by the operations (in addition to the previous condition that at least $\lfloor N/2 \rfloor$ replicas in total signal that they have accepted).

[6] For ease of integration with our implementation, we implemented a custom workload generator that uses the same distributions as YCSB via a direct translation of the YCSB code into Go.

	JP	CA	OR	VA	IRL
Japan	0.4	120	120	180	270
California		0.4	20	85	150
Oregon			0.4	75	170
Virginia				0.4	92
Ireland					0.4

Table 1: Approximate round-trip times between datacenters in milliseconds.

distribution, the most popular items differ across datacenters: the sequence of keys ordered by popularity is a different random permutation for each datacenter. We ran experiments for two workload ratios of Puts to Gets—1:1 and 1:9—with each client choosing the operation type at random. The skewed Zipf distribution is the ideal workload for quorum leases because clients in different datacenters will mostly access different objects. The uniformly distributed workload, on the other hand, is the worst case scenario for quorum leases because an object is equally likely to be accessed by replicas that hold a lease for it as by replicas that do not.

5.2 Latency

We ran this workload in the wide area, for one hundred thousand keys[7], with fifty simultaneous clients, ten at each of the five locations. Each client sends ten thousand requests to the co-located replica, in a closed loop, and measures the latency (there are thus 500,000 requests in total). For writes, clients are notified as soon as the operation has been committed, so we do not wait for it to be executed. This is sufficient to ensure strict serializability: after receiving the commit notification, the client can safely assume that any subsequent operations (proposed by itself or other clients) will be globally ordered after its write. A read, on the other hand, is executed before notifying the client, so that the read value can be returned with the notification.

In all experiments, the Multi-Paxos leader is in California. Based on the round-trip times shown in Table 1, California is the datacenter closest to the center.

We set the lease parameters as follows: the lease duration was 2 seconds, every grantor renewed the current lease after 500 milliseconds, and the lease configuration was updated every 10 seconds.

The cumulative distribution of latencies, separate for reads and writes, is presented in Figure 6 for all three lease strategies: quorum leases, single-leader lease, and Megastore-type leases. For quorum leases we run both Zipf-distributed and uniformly distributed workloads. For the skewed workload, we let the system adapt for 5000 requests from each client before we begin measuring latencies. The ratio of reads to writes is 1:1. This high frequency of writes increases the chance that reads will have to wait for concurrent writes to the same object to finish before they can execute, so, for completeness, we also present a summary of quorum lease results for the 9:1 read to write ratio in Table 2 (the other two leasing strategies benefit only marginally from this workload change, their read latencies being alread very low and very high, respectively).

The single leader lease provides the best write performance because the leader can always choose the fastest quorum for committing a write. The leader is the sole owner of the lease, so no particular other replica must be notified synchronously when an update occurs. The fastest quorum always includes the replica closest to the client, to take full advantage of the optimization described in Section 4. This low write latency, however, comes at a cost: only the leader can read locally. Thus, the single leader lease suffers mean read latency at least two orders of magnitude larger than that of the competing strategies.

The Megastore-type lease allows every replica to read any object locally, but requires proposers to

[7]A larger key space would be advantageous for quorum leases, making it less likely for a key to be accessed by a non-leaseholder.

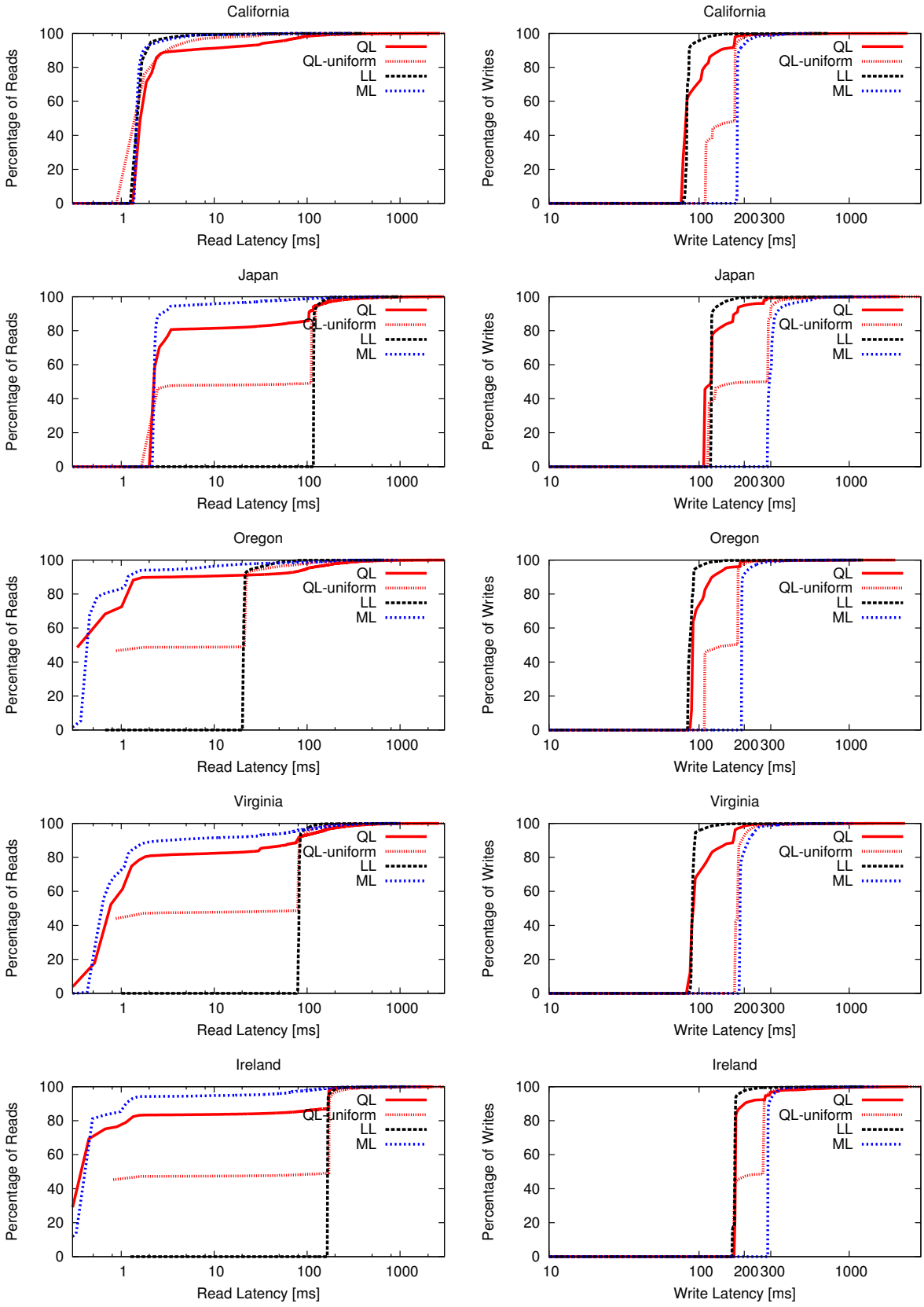

Figure 6: CDFs of client-observed latency for each site, with all three lease techniques: quorum lease (QL), single leader lease (LL), and Megastore-type lease (ML). QL-uniform corresponds to quorum leases for a uniformly-distributed workload. The read-to-write ratio in these experiments was 1:1. The Multi-Paxos leader is always located in California. Note the log scale on the X axis.

	Fast local reads
Japan	81%
California	95%
Oregon	89%
Virginia	89%
Ireland	81%

Table 2: Percentages of fast local reads ($\leq$ 10 ms) for wide-area quorum leases with 10% writes and 90% reads, Zipf-distributed.

notify *all* other replicas synchronously before committing a write. Typical (95%) read latencies are under 10 ms—one to two orders of magnitude faster than when communication with a remote datacenter is required. A small percentage of read requests are delayed by concurrent writes to the same objects: when a write is ongoing, a replica must delay interfering reads until it can be sure that the write will be committed (i.e., typically until it has received a commit notification). The price of this near-optimal read performance is increased write latency, by more than 100 ms in most cases compared to the other leasing schemes. Each replica must wait for the replica *farthest* away to acknowledge an update before committing it and notifying the client. Furthermore, this scheme incurs more risk of unavailability for writes: any unresponsive replica will prevent all updates from progressing until the replica's lease expires.

Quorum leases are a compromise between the two previous schemes. Over 80% of reads at every site are performed locally. Over 70% of updates have the minimum latency achievable by a geo-distributed Multi-Paxos system, matching that provided by the single leader lease, and 2$\times$ to 3$\times$ faster (i.e. 100 to 200 milliseconds lower latency) than writes with the Megastore approach. Because all replicas are likely to be part of at least one quorum lease, any replica failure will cause some unavailability for writes. However, this does not affect all writes (as it would for Megastore-type leases), but only writes to the objects granted to the failed replica.

The worst scenario for quorum leases is that when the workload is uniformly distributed (also presented in Figure 6). For that experiment, we set the quorum leases statically, based on geographical proximity: one lease includes the replicas in California, Japan and Oregon, the other includes California, Virginia and Ireland. Each leases half the key space. As expected, approximately half the request at each location will therefore have the minimum latency, while the other half will exhibit the worst case latency.

5.3 Throughput in a Cluster

While our focus is on wide-area replication, we evaluate the throughput of the three leasing strategies in a local-area cluster as well. Figure 7 comapres the maximum read and write throughputs achieved with all three leasing strategies. For quorum leases, we present results for two situations: (1) different objects are popular at different replicas—i.e., when trying to read a certain object, all clients know which replica has a lease for that object; and (2) clients are oblivious of lease assignment, and direct their reads uniformly at random across all replicas.

Megastore leases have the best read throughput, narrowly surpassing quorum leases when clients are aware of lease assignments, and 4$\times$ higher than single-leader leases. Because we use batching to commit writes (the leader batches up to 5000 updates at a time), the difference in messaging patterns between the three lease implementations is less important, so their write throughputs are approximately the same. Quorum leases have a 10% lower write throughput because we must separate the updates into per-quorum batches (i.e., only updates leased by the same quorum are batched together). In the local cluster, Megastore leases achieve higher throughput, at the cost of suffering more impact upon node failure: *all* leases will be stalled when any replica becomes available, whereas with a quorum lease, an unresponsive replica will

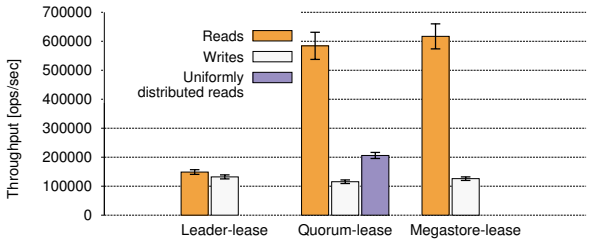

Figure 7: Local-area read and write throughput for different leasing strategies. The "Uniformly-distributed reads" for quorum leases corresponds to the situation when clients do not know which replicas can read locally which objects. Error bars represent 95% confidence intervals.

stall only updates for the objects it has leased. In general, our results suggest that the differences between Megastore-type leases and quorum leases are less important in a local cluster than they are in the wide area, if throughput is the main consideration.

5.4 Discussion

These experiments single-object operations. For multi-object operations, if the same objects are usually accessed together, then they will be leased together, and therefore the corresponding read and write latencies will be similar to those reported here. On the other hand, the write latency of multi-object writes that target objects leased by different quorums will approach that of the Megastore leases. If such operations are common, and multi-object reads that span quorums are also common, the Megastore lease may be a more suitable.

6 Related Work

Time-based leases have been introduced as a way of improving the latency of reads while maintaining strong data consistency in distributed systems [2, 4, 11, 13, 24]. Quorum leases preserve this goal in the context of Paxos replicated state machines.

Previous systems have used leases to improve the read performance of Paxos systems in two ways. First, Chandra et al. [6] proposed the Paxos leader lease (also used in Chubby [4] and Spanner [8]), which gives the Multi-Paxos leader the ability to perform strongly-consistent reads locally, for a specified time interval. Second, Megastore [3], a wide-area deployment of Paxos, effectively grants every replica in the system a lease for every object: a write in Megastore must synchronously send an invalidate message to every replica that has not accepted the write—so that the remote replica knows its copy of the object (*entity group*, in Megastore terminology) is stale. As shown in our evaluation, the leader lease and the Megastore lease are at opposite extremes of the design spectrum for leases in Paxos: leader leases allow for optimal Multi-Paxos write latency, while Megastore leases sacrifice write latency for optimal read latency at every replica. By contrast, quorum leases allow for a more fine-grained exploration of the design space: leases can be granted to any subset of replicas and they refer to only a specified portion of the replicated state, so that different replicas can hold different leases at the same time. For a geo-distributed replicated state machine that observes locality in the popularity of its replicated objects, quorum leases can approximate the benefits of both previous approaches simultaneously. This comes at the expense of simplicity, because quorum leases are more complex to manage.

Spanner [8] is a leader-leased, Paxos-based system that uses TrueTime—accurate clock synchronization clock synchronization that requires special hardware—to improve geo-distributed read performance. It

does so in two ways: first, it improves the management of leader leases, by taking advantage of their synchronized clocks. Second, it can let remote replicas read locally by issuing *snapshot reads*, at timestamps speficied by clients. Like a quorum lease, a snapshot read executes locally. Unlike a quorum lease, however, a snapshot read may not be up-to-date with respect to updates already committed at other replicas.

Other systems, such as ZooKeeper [14] and MDCC [17], allow fast local reads, but make no freshness guarantees (i.e., the results may be stale).

Previous systems have used Paxos as a mechanism for granting leases: FaTLease [15] and PaxosLease [23] are replicated state machines that grant leases to individual clients. They use Paxos to ensure the fault tolerance of the lease-management system, and take advantage of the perishable property of leases to avoid logging Paxos state to disk. By contrast, quorum leases are granted to the replicas themselves (not the clients) based on the popularity of replicated objects, and they can be granted to multiple entities (replicas) at the same time instead of just one. Furthermore, unlike FaTLease and PaxosLease, quorum leases specify how leases are enforced, not just how they are granted.

Although we have presented quorum leases only in the context of Multi-Paxos, the most widely used variant of Paxos, we believe they can also be applied to other Paxos variants [20, 21, 22], as well as systems using similar majority-consensus replication protocols [14]. Leaderless Paxos variants, in particular, like Mencius [21] and Egalitarian Paxos [22] are a good fit for quorum leases because they do not require a single replica to be part of every write quorum—this gives more flexibility in choosing lease quorums, and benefits both write latency and availability, especially with Egalitarian Paxos, which optimizes for wide-area Paxos commit latency.

The use of leases in improving the performance of distributed protocols is not restricted to Paxos. Zzyzx [12] is a Byzantine fault tolerant system that gives clients exclusive locks to objects. It does so in order to preclude competing client requests, and thus to reduce the common commit path by one message delay (half a round trip) when compared to Zyzzyva [16]. Compared to the mechanism in Zzyzx, quorum leases are granted to the replicas themselves, so multiple (or all) clients can benefit from each lease, and solve a different problem: allowing single-replica consistent reads, an operation generally incompatible with Byzantine fault tolerant systems.

Per-client leases are also used in systems such as Chubby [4] and Farsite [2] to allow clients to operate on cached sub-parts of the replicated data (i.e., cached files).

Quorum leases are superficially similar to the notion of *preferred quorums* [1, 9]. The key distinction is that a quorum lease constitutes a requirement to synchronously update each lease holder, while preferred quorums are a performance optimization. In Q/U [1], clients try to communicate with the same preferred quorum of replicas every time they want to access a particular object, to increase the chance that those replicas are up-to-date with all the latest versions. Quorum leases, by contrast, *guarantee* that replicas in the lease holder subset are up-to-date, so clients can read from any one replica in that subset and not an entire quorum. HQ replication [9] makes only a quorum of replicas execute the full protocol in failure-free situations to reduce the messaging overhead.

7 Conclusion

This paper presented the design and implementation of quorum leases for Paxos. By exploiting the existing messaging requirements for Paxos operations, quorum leases provide a natural, and therefore efficient, mechanism for allowing local reads in a Paxos-replicated system without substantially increasing the delay for write operations to commit. Our evaluation on geo-distributed replicas in Amazon EC2 datacenters shows that these leases work well in practice: More nodes can perform reads locally than with simple master-only leases, but the write latency increases only modestly and only does so for typically fewer than 10-20% of operations. We therefore believe that quorum leases are an excellent general-purpose leasing

mechanism for Paxos-based systems.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–74, Brighton, UK, October 2005.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. 5th USENIX OSDI*, pages 1–14, Boston, MA, December 2002.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[4] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. 7th USENIX OSDI*, Seattle, WA, November 2006.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[6] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proc. 26th ACM SOSP*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[7] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX OSDI*. USENIX, 2012.

[9] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. 7th USENIX OSDI*, pages 177–190, Seattle, WA, November 2006.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[11] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

[12] J. Hendricks, S. Sinnamohideen, G.R. Ganger, and M.K. Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 363–372, 2010.

[13] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[14] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*, USENIXATC'10, Berkeley, CA, USA, 2010. USENIX Association.

[15] Felix Hupfeld, Björn Kolbeck, Jan Stender, Mikael Högqvist, Toni Cortes, Jonathan Marti, and Jesús Malo. Fatlease: scalable fault-tolerant lease negotiation with paxos. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 1–10, 2008.

[16] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, Stevenson, WA, October 2007.

[17] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proc. 8th ACM European Conference on Computer Systems (EuroSys)*, April 2013.

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

[19] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), December 2001.

[20] Leslie Lamport. Fast Paxos. http://research.microsoft.com/apps/pubs/default.aspx?id=64624, 2006.

[21] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proc. 8th USENIX OSDI*, pages 369–384, San Diego, CA, December 2008.

[22] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[23] Marton Trencseni, Attila Gazso, and Holger Reinhardt. PaxosLease: Diskless Paxos for Leases. http://arxiv.org/pdf/1209.4187.pdf, 2012.

[24] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):563–576, July 1999.