GigaPaxos: Service Dispersability via Group-Scalable, Reconfigurable Replica Coordination

Abstract

We present GigaPaxos, a novel system for group-scalable replica coordination. Group scalability refers to the ability to easily manage a very large number of separate and quickly reconfigurable replica groups, one for each lightweight fault-tolerant principal as small as a single record in a key-value store or an ephemeral service replica created on the fly each end-user. GigaPaxos achieves this goal by driving down the marginal memory overhead of a replicated state machine consensus group to a few hundred bytes while keeping the messaging overhead, throughput, and latency of each group independent of the total number of groups and comparable to or vastly better than state-of-the-art consensus systems. Our comparative evaluation shows that state-of-the-art consensus can only sustain up to tens or hundreds of consensus groups on commodity machines while GigaPaxos can easily support millions. We study the benefits of group scalability using several case studies including myCloud, a hypothetical cloud application that creates a custom, reconfigurable replica group for each user’s personal cloud data, and show that agile group scalability can significantly enhance user-perceived performance.

1 Introduction

Many large-scale distributed systems use a combination of replication and partitioning of data and computation for balancing several different objectives such as fault-tolerance, performance, scalability, resource cost, or ease of management. These competing objectives are often conflicting, e.g., increasing the degree of replication across failure-independent machines improves availability but increases the overhead by a proportional or worse factor. Thus, a common approach is to partition the overall system, i.e., the state and the associated computation, into smaller distributed principals spread across different subsets of machines, thereby improving performance by increasing concurrency; fault isolation across partitions placed on fault-independent sets of machines; overall capacity by not restricting it to a single machine’s capacity; and manageability via the flexibility to independently provision resources across partitions.

Figure 1: Replication and partitioning combinations.

Figure 1 shows increasingly finer-grained replication and partitioning schemes. Compared to the full replication baseline (e.g., replicated state machine based systems [27, 16, 11]), sharding or partitioning principals across non-overlapping machines significantly improves overall capacity provided the system’s consistency semantics allow operations on different shards to proceed in parallel. Figure 1(d) shows the most general combination of replication and partitioning wherein different partitions may not necessarily be replicated on non-overlapping machines, a popular approach today [15, 17, 13, 32]. This option further improves performance by increasing concurrency when different replica groups isolate principals into independent consistency groups. Indeed, simply increasing the logical number of partitions even with full replication can improve overall system capacity, as shown in 1(c).

Our position is that these traditional distributed system designs are monolithic, i.e., they stop well short of the finest achievable grain of repliconfigurability, informally (formalized in §4.3) the flexibility to assign different subsets of machines to manage fine-grained distributed principals and quickly reconfigure this assignment. The extent of repliconfigurability permitted by a design, as argued in §4.3, has non-trivial operational implications. For example, imagine a user of myCloud—a hypotheti-
cal personal cloud application that also forms one of our prototype-driven end-to-end case studies (§4.3.1)—who expects her personal cloud data like documents, calendar, mail, media metadata, etc. to be readily available right at or close to the (virtualized) access points across which she zips by in her hyper-mobile always-connected life; or imagine her editing a cloud-based shared document collaboratively with a colocated co-worker. In such use cases of “edge clouds” (also variously referred to as cloudlets, multi-tenant gateways, micro-clouds, etc.), caching of static content alone is insufficient or irrelevant because of consistency required of mutable data; instead, they need system support for agile repliconfigurability, in line with the vision of “fluid replication” from the 90s.

Our contribution, GigaPaxos, is a small but concrete step towards realizing the fluid replication vision. We envision applications “sprinkling” principals as small as a single record in a key-value store, a counter, a user’s calendar, a shared document, etc. wherever and whenever needed without sacrificing consistency. A key challenge that GigaPaxos addresses is group scalability, i.e., the ability to scale to a very large number (millions or more) of independent consensus groups. Consensus is at the heart of most distributed systems and (consensus-based) replicated state machines implement linearizability, the strongest form of consistency, while also acting as an important building block for replica coordination schemes with weaker consistency (refer §3.1). Although a large body of prior work has focused on improving the performance, cost, or robustness of consensus-based systems, group scalability is a dimension that appears to have not been explored before. Indeed, we find that state-of-the-art Paxos or other consensus implementations can barely sustain tens or hundreds of groups. In comparison, GigaPaxos can scale to millions of Paxos groups on commodity machines with little performance or cost penalty.

GigaPaxos achieves group scalability through a novel design and implementation that carefully separates idle and active Paxos groups so as to drive down the memory overhead of an idle Paxos instance to a few hundred bytes; uses a novel hot-swap technique to pause idle Paxos instances; amortizes the overhead of failure detection and logging across groups; enables programmatic policy for automating group reconfiguration at scale; and uses a highly event-driven design that does not rely on any per-instance background tasks that are commonplace in consensus implementations (refer §3.1).

We have implemented a prototype of GigaPaxos with a simple API that allows any “black-box” application, even those not originally designed with fault-tolerance or replication in mind, to leverage repliconfigurability. Our prototype-driven experiments show that:

1. GigaPaxos achieves comparable or vastly superior performance compared to state-of-the-art consensus implementations even for a single group but comfortably scales to orders of magnitude more groups (§4.1).
2. GigaPaxos’ replicable API and support for programmatic reconfiguration policies are easy to use with a number of third-party applications (§4.3).
3. Per-object reconfigurability can significantly enhance end-to-end client-perceived performance for massively geo-distributed edge cloud services (§4.3).

2 Case for repliconfigurability

A founding position of our work is that repliconfigurability, informally the flexibility to assign different subsets of machines to manage different groups of replicated principals (objects or services) is a valuable abstraction in large-scale distributed systems. We inspect many recent and classical distributed systems in order to make qualitative and quantitative arguments to support this position.

Consistency vs. concurrency: Any system managing distributed state must reckon with consistency requirements. Any consistency semantics, including weak or eventual consistency, necessitates imposing some restriction on the ordering of or isolation across operations accessing that state, i.e., ensuring consistency fundamentally reduces the concurrency permitted by the system and consequently the throughput capacity of the system. (Note that this claim is true even of undistributed systems, e.g., a single-machine database with atomicity, consistency, isolation, and durability (ACID) requirements, but is even more so in distributed systems.) Object-group configurability improves performance by allowing operations across independent sets of objects to proceed concurrently.

Availability vs. overhead: High availability entails larger replica groups (with proportional or worse overhead) until the marginal benefit of increasing the replication factor is outweighed by its overhead, at which point it is more effective to partition the data across non-overlapping groups of machines. Partitioning data across non-overlapping replica groups is a special case of repliconfigurability.

Scalability vs. manageability: Any scalable distributed system design must choose between two conflicting goals: on one extreme are randomization-based approaches (leftmost in in Figure 2) that are simple and scale beautifully with no single point of failure or congestion; on the other extreme are increasingly planned approaches that are easier to manage offering superior fault isolation and the flexibility to provision resources or tailor usage to individual user preferences.

To concretize the high-level exposition above, let’s study these tradeoffs in the context of several modern large-scale distributed systems as shown in Figures 2(a)–2(c) that show that the extent of repliconfigurability allowed by a system’s design has nontrivial operational im-
applications, as illustrated in Figures 2(a)–2(c). A baseline example with little flexibility is consistent hashing with replication, e.g., Amazon Dynamo is a key-value system that uses consistent hashing to determine the replica group of machines that manage an object. While this approach is simple and scales well when machines and object workload patterns exhibit predictable characteristics, it is cumbersome in environments with more unpredictability and flux such as peer-to-peer (P2P) environments. Sharing this motivation, Scatter [17], a P2P storage system uses consistent, configurable groups as a defining abstraction. Scatter’s support for “amoebic” reconfiguration of groups, i.e., the ability to split, merge, or migrate members of objects across adjacent consensus groups, enhances a group’s ability to self-organize under dynamic conditions while maintaining linearizability consistency for operations to a single object.

A system like Google’s Spanner [13] significantly increases repliconfigurability over static or amoebic replica groups. Spanner has a fixed number of pre-defined (or slow-changing) Paxos groups to which it maps a large number of directory objects (i.e., a bag of key-value mappings) in a many-to-one manner. Spanner allows administrators to control the “number and types of replicas, and the geographic placement of those replicas”, for example, by specifying policies such as [Object A: North America, replicated 5 ways with 1 witness]; [Object B: Europe, replicated 3 ways], etc. However, Spanner is designed to remap objects across existing Paxos groups, not reconfigure the Paxos groups themselves. The distinction is important as the total number of conceivable consensus groups is exponential in the total number of machines, so a practical system is forced to create a manageable1 number of packaged groups (like N. America, Europe, etc.) and adopt a many-to-one principal-to-group mapping, an approach that works well in the common case.

Our goal is to take repliconfigurability to the extreme, namely, allow for each arbitrarily small object to be mapped to an arbitrary consensus group specifically for that object. We refer to this flexibility as maximal repliconfigurability, wherein repliconfigurability is defined as the ratio of the total number of objects to the total number of separate consensus groups in the system. Thus, the maximal value is 1; for Spanner, it is typically much lower than 1; for Scatter or Dynamo, it is roughly equal to the ratio of the number of machines and the product of the number of keys and the average replication factor.

Our vision is similar to that of fluid replication [30] proposed by Noble et al. in the late 90s or more recently referred to as “dispersable computing” [14]. Our goal of agile reconfigurability also overlaps with more recent systems like Tuba [8] but differs significantly in its focus on group scalability and the powerful RSM abstraction.

This paper requires the reader to be familiar with Paxos[21]. The appendix has a brief primer, and [22] and [33] are good resources respectively for a simplified conceptual and implementation-oriented exposition.

3 GigaPaxos design

GigaPaxos is designed to meet the following goals.

(1) Agile repliconfigurability: An application should be able to easily request or change a consensus group for a fine-grained fault-tolerant object.

(2) Group scalability: The aggregate performance (capacity and latency) across consensus groups should be independent of the total number of consensus groups.

(3) Application agnosticism: The design must provide a simple API for black-box applications, remaining agnostic to application-specific details.

(4) Automated reconfiguration: Applications should be able to specify policies to programmatically reconfigure the membership of the consensus instances.

(5) Control plane scalability: There must be no single point of congestion or failure including the control plane managing dispersion of distributed principals.

3.1 Design overview

To address the above goals, GigaPaxos is designed as a two-tier reconfigurable consensus engine consisting of two logically distinct types of nodes: app-containers and reconfigurators. A group of app-containers form a consensus group for a named object that they manage. A group of reconfigurators form a consensus group that is responsible for making decisions about when and how to reconfigure the app-container group for a subset of objects, and to help correctly redirect client requests to the
current group. An app-container encapsulates a third-party application that contains the logic needed to process a client request, modify the corresponding object state, and send a reply back to the client.

Figure 3: GigaPaxos group-scalable architecture combining randomization and planned placement benefits.

Figure 3 shows reconfigurators on the left organized as a consistent hash ring with a fixed number of clockwise contiguous nodes on the ring forming a consensus group to manage all principals mapping to the first node. Unlike traditional consistent hashing based schemes however, reconfigurators only maintain directory information about app-container nodes managing each principal. For example, the figure shows principal X being managed by reconfigurators 12, 3, and 7 and the corresponding application replica group being maintained on app-containers 3, 4, 6, 7. Lookup requests to reconfigurators are expected to be infrequent as clients can opportunistically cache this information until it becomes stale.

Applications can specify if only a subset of request types need replica coordination, allowing them to use consensus as a building block for different consistency semantics. For example, enforcing consensus for every client request to a named object ensures linearizability (as in [17]) across all operations to that object while enforcing consensus for writes alone (or reads alone) ensures sequential consistency for all operations to that object [9]. Relaxing it further to eventual consistency does not need consensus among replicas, but reconfigurators must still rely on consensus to make reconfiguration decisions in a fault-tolerant and consistent manner (§3.6.1).

GigaPaxos as described has no single point of failure or congestion (design goal #5). Reconfigurators form the control plane and consistent hashing with replication ensures availability and load balance. Reconfigurators are homogeneous as they perform quick predictable control plane tasks. The separation of reconfigurators and app-containers combines the best of randomization and planned placement. Reconfigurators periodically receive demand reports from app-containers and, based on a configurable principal-specific policy, reconfigure app-container replica group managing the principal.

We describe how GigaPaxos achieves its remaining goals using the following key mechanisms described in the following subsections: (1) a compact representation of Paxos instances; (2) separating and amortizing machine-specific overhead from group-specific overhead; (3) a hot-swap mechanism to relieve memory pressure while maintaining correctness; (4) a group-scalable persistent logger; and (5) simple client API and programmatic reconfigurability support.

The following terms are used throughout the paper: a Paxos instance is the Paxos-related, application-agnostic state stored at a machine for a single, named object; a Paxos group is the set of distributed Paxos instances managing a single object, which in conjunction with the application logic forms the corresponding RSM.

3.2 Managing compact Paxos instances

GigaPaxos’ core consists of a PaxosManager per machine that is responsible for machine-specific functions of which there are four key ones: (1) Paxos instance management, (2) persistent logging, (2) failure detection, and (3) messaging and demultiplexing.

PaxosManager maintains a map from the name of a Paxos group, objectID, to a data structure maintaining the minimum Paxos instance state necessary for safety, i.e., the state blocks marked “Fixed Instance”, “Acceptor Idle”, and “Coordinator Idle” respectively in Fig. 4. The first remains unchanged throughout the lifetime of this Paxos instance, i.e., until the epoch and group are reconfigured or the object is deleted. The latter two are referred to as idle state because this state must be remembered by each Paxos group member even during periods when the group is not actively processing client requests.

3.2.1 Idle Paxos instance state

GigaPaxos as described has no single point of failure or congestion (design goal #5). Reconfigurators form the control plane and consistent hashing with replication ensures availability and load balance. Reconfigurators are homogeneous as they perform quick predictable control plane tasks. The separation of reconfigurators and app-containers combines the best of randomization and planned placement. Reconfigurators periodically receive demand reports from app-containers and, based on a configurable principal-specific policy, reconfigure app-container replica group managing the principal.

We describe how GigaPaxos achieves its remaining goals using the following key mechanisms described in the following subsections: (1) a compact representation of Paxos instances; (2) separating and amortizing machine-specific overhead from group-specific overhead; (3) a hot-swap mechanism to relieve memory pressure while maintaining correctness; (4) a group-scalable persistent logger; and (5) simple client API and programmatic reconfigurability support.

The following terms are used throughout the paper: a Paxos instance is the Paxos-related, application-agnostic state stored at a machine for a single, named object; a Paxos group is the set of distributed Paxos instances managing a single object, which in conjunction with the application logic forms the corresponding RSM.

3.2 Managing compact Paxos instances

GigaPaxos’ core consists of a PaxosManager per machine that is responsible for machine-specific functions of which there are four key ones: (1) Paxos instance management, (2) persistent logging, (2) failure detection, and (3) messaging and demultiplexing.

PaxosManager maintains a map from the name of a Paxos group, objectID, to a data structure maintaining the minimum Paxos instance state necessary for safety, i.e., the state blocks marked “Fixed Instance”, “Acceptor Idle”, and “Coordinator Idle” respectively in Fig. 4. The first remains unchanged throughout the lifetime of this Paxos instance, i.e., until the epoch and group are reconfigured or the object is deleted. The latter two are referred to as idle state because this state must be remembered by each Paxos group member even during periods when the group is not actively processing client requests.

3.2.1 Idle Paxos instance state

GigaPaxos as described has no single point of failure or congestion (design goal #5). Reconfigurators form the control plane and consistent hashing with replication ensures availability and load balance. Reconfigurators are homogeneous as they perform quick predictable control plane tasks. The separation of reconfigurators and app-containers combines the best of randomization and planned placement. Reconfigurators periodically receive demand reports from app-containers and, based on a configurable principal-specific policy, reconfigure app-container replica group managing the principal.

We describe how GigaPaxos achieves its remaining goals using the following key mechanisms described in the following subsections: (1) a compact representation of Paxos instances; (2) separating and amortizing machine-specific overhead from group-specific overhead; (3) a hot-swap mechanism to relieve memory pressure while maintaining correctness; (4) a group-scalable persistent logger; and (5) simple client API and programmatic reconfigurability support.

The following terms are used throughout the paper: a Paxos instance is the Paxos-related, application-agnostic state stored at a machine for a single, named object; a Paxos group is the set of distributed Paxos instances managing a single object, which in conjunction with the application logic forms the corresponding RSM.

3.2 Managing compact Paxos instances

GigaPaxos’ core consists of a PaxosManager per machine that is responsible for machine-specific functions of which there are four key ones: (1) Paxos instance management, (2) persistent logging, (2) failure detection, and (3) messaging and demultiplexing.

PaxosManager maintains a map from the name of a Paxos group, objectID, to a data structure maintaining the minimum Paxos instance state necessary for safety, i.e., the state blocks marked “Fixed Instance”, “Acceptor Idle”, and “Coordinator Idle” respectively in Fig. 4. The first remains unchanged throughout the lifetime of this Paxos instance, i.e., until the epoch and group are reconfigured or the object is deleted. The latter two are referred to as idle state because this state must be remembered by each Paxos group member even during periods when the group is not actively processing client requests.

3.2.1 Idle Paxos instance state

GigaPaxos as described has no single point of failure or congestion (design goal #5). Reconfigurators form the control plane and consistent hashing with replication ensures availability and load balance. Reconfigurators are homogeneous as they perform quick predictable control plane tasks. The separation of reconfigurators and app-containers combines the best of randomization and planned placement. Reconfigurators periodically receive demand reports from app-containers and, based on a configurable principal-specific policy, reconfigure app-container replica group managing the principal.
has been stopped, which is needed to perform reconfiguration safely (§3.6.1).

A coordinator, strictly speaking, does not have to persistently maintain any idle state at all as coordinators are already presumed to be perishable. However, garbage collecting coordinator state during idle periods means that a new coordinator must be elected (with the first PREPARE phase) upon the arrival of a client request. In order to maintain Paxos’ low, essentially optimal, message overhead per client request during graceful execution, i.e., just the second ACCEPT/DECISION phase, it is important to support long-lived coordinators. So, each GigaPaxos coordinator instance must either maintain all of the coordinator idle state in Fig. 5 or immediately relinquish its role as coordinator by ceasing to commandeer further proposals in its ballot.

The coordinator’s idle state must thus maintain: (1) nextProposalSlot, the lowest slot such that the coordinator has not yet used that or any higher slots to commandeer any proposals; (2) its ballot that in general may be out of sync with the local acceptor’s perceived ballot; (3) isElected, indicating whether its ballot has been accepted by a majority of acceptors, at which point it can garbage-collect its pre-election state (§3.2.2); and (4) memberFrontiers, the slot numbers up to which, in its view, acceptors have cumulatively executed application requests; the coordinator piggybacks the median slot number in its ACCEPT and DECISION messages to all acceptors who use it to refresh their majorityFrontier.

Compactness. The point of listing the seemingly mundane details above is to emphasize that this state—the variables in the three shaded boxes in Fig. 4 plus the connecting pointers—is literally all of the state GigaPaxos adds per idle Paxos instance to whatever state the application itself maintains. The size of this idle state is ≈350 bytes for Paxos instances with three members in our implementation; larger groups cost 8 more bytes (or one integer each in the two int arrays).

### 3.2.2 Active Paxos instance state

An active Paxos instance, i.e., one that is currently agreeing on the order of client requests for the underlying application, typically needs to maintain much more state than the idle state above. Fig. 5 illustrates the active state that must be maintained for safety.

An acceptor’s active state consists of (1) a sequence of accepted proposals in slot number order, possibly with gaps, that it has previously accepted, and (2) a set of committed decisions received out-of-order. The former sequence starts at majorityFrontier+1 or higher, and the latter sequence starts strictly higher than nextProposalSlot, the first slot for which no decision has been received.

A coordinator’s active state additionally consists of (1) adopted proposals, i.e., lower-ballot proposals received from acceptors in their replies to this coordinator’s PREPARE message, wherein the coordinator picks for each slot the proposal with the highest ballot; (2) one waitFor data structure (not shown) to track whether a majority of acceptors have replied successfully to the PREPARE message; and (3) myProposals, a sequence of proposals being commandeer by the coordinator, i.e., proposals for which it has or will send out ACCEPT messages, and for each of which it maintains a waitFor structure to track a majority of acceptances. The first two are needed only until the coordinator gets elected by receiving a majority of suppo PREPARE replies. If a coordinator receives any client requests during this election, it enqueues them with the first available (tentative) slot number in myProposals. When a coordinator receives a PREPARE majority (“view change” in Fig. 5), it merges all of the adopted proposals into and with strict priority over myProposals, marks itself as active, and begins commandeer myProposals. An active coordinator thus only maintains a single queue, myProposals, of proposals awaiting majority acceptance; when that happens, they are announced as committed decisions to all acceptors and are dequeued.

Bulk. The size of an active Paxos instance can be orders of magnitude larger than an idle Paxos instance, e.g., a burst of rapid requests to a group can result in thousands [18] of requests being concurrently processed, each causing hundreds or thousands of bytes of queue entries at acceptors as well as coordinators, thereby easily inducing megabytes of state. This active state needs to be maintained at an acceptor until a majority of acceptors have caught up, i.e., majorityFrontier+1 equals nextProposalSlot, and at a coordinator until it is no longer commandeer any proposals, i.e., myProposals is empty.

### 3.3 Bounded number of active instances

We claim that under realistic conditions, with a very large number of consensus instances, the number of idle instances will overwhelmingly dominate active ones. This insight motivates GigaPaxos’ hot-swap mechanism.

Consider a GigaPaxos application distributed across $M$ machines managing a total of $N$ objects with each object managed by a separate consensus group. Let $T$ denote the average response time of a request with state machine
replication, inclusive of both the unreplicated application execution time and the latency to establish its consensus order. Suppose the maximum request throughput that can be steadily sustained by the underlying (unreplicated) application on a single machine is \( C \) per second. By Little’s law [25], the average number of outstanding requests being processed at any single machine is \( A = C \cdot T \). Note that, if \( C \) and \( T \) are fixed, \( A \) is independent of the size of a consensus group, the total number of machines \( M \), or the total number \( N \) of objects in the system.

For example, if \( C \) is 25,000 requests/sec and the average response time of a request is as high as \( T = 500 \) ms, then the average number of outstanding requests at a machine is 12,500. In practice, the throughput of most applications employing an RSM approach is likely to be much lower, e.g., for a database application, synchronous random write throughput is typically on the order of a hundred/sec with hard drives, and up to several thousand/s/sec with typical solid state drives.

The number of active consensus instances at a machine is at most the total number of outstanding requests being processed at that machine. Indeed, the worst case workload is one that, in a round-robin manner, issues requests to all other objects (or consensus groups) before returning to the first. Thus, in a GigaPaxos system with millions of consensus groups, the vast majority of consensus instances must be idle.

There are two caveats however: (1) this analysis implicitly assumes graceful or failure-free execution; (2) even if the average size of an idle consensus instance is small the total number of Paxos groups that can fit in memory on commodity hardware is limited, e.g., with 16GB memory and 400 bytes per Paxos instance, the number of sustainable idle instances is 40 million. To address these issues, GigaPaxos uses hot swapping, a mechanism that helps GigaPaxos scale to billions of groups per machine with commodity disk capacities.

### 3.3.1 Hot swapping Paxos instances

A simple hack to juggle too many Paxos instances on a machine is for the manager to simply “soft-crash” that Paxos instance, i.e., to dequeue it from its instances map allowing for the state get garbage collected. This action will preserve safety as it will just appear to the rest of its group like a member failure. However, this simplistic approach has several shortcomings. First, it forces a roll forward of the Paxos instance from the most recent checkpoint when a request for a Paxos group arrives at a manager, stalling the request handling until the recovery is complete. The alternative of simply not handling the request is not viable, as that will over time prevent most Paxos groups from making any progress at all, a much worse state of affairs than the theoretical lack of guarantee of liveness under asynchrony. Second, the overhead of doing a checkpoint recovery upon a request arrival as a common case operation can itself overwhelm memory, computation, and I/O cycles on a machine severely hurting overall performance.

GigaPaxos instead employs a far nimbler hot swapping technique that capitalizes on the two observations above: (1) most Paxos instances will be idle when the total number of instances on a machine is very large; and (2) idle state is extremely compact (Fig. 4 as opposed to 5). To this end, the manager on each machine maintains a background process that periodically but infrequently (e.g., every few minutes), makes a sweep over all active instances and pauses instances that have been idle for the threshold interval, i.e., it synchronously dequeues the instance from its map and writes the compact idle state to a database. Subsequently, upon the arrival of a client request or a Paxos protocol message for that instance, the manager’s demultiplexer as usual first consults its instance map to route the message. If the instance is not found, the manager must check the database for paused state that, if found, must be used to reconstruct the Paxos instance. Hot swapping shares some similarities with Cheap Paxos [24] or ZZ [34] for bringing up virtual machines, but those approaches are comparable to the “crash” option above.

A downside of hot swapping is that it imposes a small latency penalty (<10ms typically) for the unpause operation. However, this penalty only impacts the first client request (or Paxos protocol message) in a burst of activity for that group. Subsequent requests do not incur any penalty as the instance will not be re-paused until it has been idle for the threshold duration. On the flip side, hot swapping will disproportionately affect unpopular Paxos instances with longer-than-threshold idle periods between successive client requests. Still, we believe that the penalty—an additional database lookup for a small record—is unlikely to significantly impact most applications as (1) most applications using consensus are likely to touch the disk for common operations anyway; and (2) with persistent logging, enabled by default in GigaPaxos, each client request must encounter at least one synchronous disk write in order for acceptors to log an ACCEPT message before responding. Finally, in geodistributed scenarios, the unpause penalty is unlikely to affect end-to-end latency as that is dominated by network delays fundamental to Paxos.

### 3.3.2 Graceful vs. failure-prone operation

With machine failures, the fraction of active instances at GigaPaxos machines can be higher. The reason is that a Paxos instance can not fully garbage-collect the log of accepted proposals at an acceptor as that requires a majority of replicas in the group to have executed (or persistently logged the corresponding decision) the ap-
plication up to that slot. Nevertheless, during periods of synchrony when at least a majority of replicas in all groups are available—exactly when Paxos guarantees liveness—healthy machines will be unaffected and only see a small number (as quantified above) of active instances. Fate sharing makes the number of active Paxos instances at failed machines a non-issue.

However, under more severe machine failure patterns that result in a significant fraction of Paxos instances on a machine being unable to make progress because of a lack of a quorum in their respective groups, the number of active instances on otherwise healthy machines can grow to unsustainable levels. There are several reasonable ways to handle this case: (1) the strawman outlined above that crashes an instance to pause it; (2) checkpointing immediately at $\text{nextSlot}$ and then crashing the instance so as to reduce the length of the roll forward; (3) pausing and unpausing active state (that could be potentially much larger than the compact idle state). All options incur higher overhead compared to hot swapping idle instances, but will not impact client-perceived latency as they are required only when the corresponding Paxos group is not live anyway. Our current implementation supports the second option.

3.4 Amortized fault detection and logging

Failure detection is a key component of any consensus implementation. Although failure detection need not be reliable (a problem as hard as consensus itself [12]), it is needs to be responsive in order to ensure prompt replacement of a failed coordinator. Failure detectors are typically implemented using keep-alives between all or a nontrivial subset of machine pairs in a consensus group. However, unlike typical Paxos implementations, group scalability in GigaPaxos makes it impractical to maintain a separate failure detector per group; for example, 1000 groups each of size 5 and a keep-alive frequency of 4 secs imply 1000 packets/sec for failure detection; with 100K groups, failure detection alone becomes a full-time job! Thus, GigaPaxos pushes failure detection to PaxosManager maintaining just one failure detector per machine as opposed to one per group.

Likewise, the persistent logger resides in the manager and is common across all Paxos instances on the machine. This design not only amortizes the overhead of logging $\text{PREPARE/ACCEPT/DECISION}$ messages across all instances, but also allows log messages from different Paxos instances to be batched, driving down the overhead of persistent logging to negligible levels. Without such batching, GigaPaxos’ request throughput will be limited by the synchronous disk write throughput.

3.5 Log indexing, pruning, compaction

In a traditional RSM, garbage collecting safety-critical acceptor logs is easy; they can simply be tail-pruned below the highest slot, $\text{majorityFrontier}$, up to which a majority (or even just $f+1$ if at most $f$ can fail) have received all decisions. This just requires tracking file offsets on disk or maintaining slot-indexed records in a database such that it is easy to check whether all logs before some offset are below $\text{majorityFrontier}$. Looking up logged messages when needed is efficient as the number of log messages is at most the checkpoint interval.

Figure 6: GigaPaxos’ group-scalable logger (bottom) compared to traditional RSM logger (top).

**Indexing.** In GigaPaxos, this indexing problem is harder. As shown in Figure 7, a single write-ahead log for all groups is extremely efficient (e.g., disabling logging improves capacity by barely 15%), but makes it difficult to track where what is logged; for example, upon a coordinator change or a catch-up request from a lagging acceptor for a group X, an acceptor needs to retrieve logged $\text{ACCEPTs}$ or $\text{DECISIONs}$ for X in a specific slot range. To this end, GigaPaxos needs to additionally maintain a log index map keyed by group names that tracks the $[\text{file,offset,length}]$ and $[\text{slot,ballot}]$ information for every logged message. This is tricky because, by design, the number of groups can be much larger than that can be stored in memory, and simply using a traditional database (even with batching) makes the critical path about two orders of magnitude slower.

**Pruning.** GigaPaxos’s log index is a swappable in-memory map that is as fast as a hash table lookup for working sets that fit in memory, but swaps infrequently used records to a database table indexed by the group name. The log itself is split across logically times-tamped files each of a fixed maximum size, and each log index record in addition to the information above tracks $\text{minLogfile}$, the log file storing that group’s log message with the lowest slot number, i.e., the lower of $\text{majorityFrontier}$ (for $\text{ACCEPTs}$) and the most recent check-
pointed slot (for **decisions**). The garbage collector periodically queries the database for the minLogfile frontier, i.e., the set of minLogfiles across all groups, and then removes log files older than the oldest log file in that frontier set from the file system.

**Compaction.** Alas, the logger’s garbage collection woes do not end here. With highly skewed workloads, for example, one where most requests go to just one (or a small number) of group(s) but a request occasionally goes to a “rare” group, it is possible that every log file contains at least one (or a few) log message that prevents the log file from being safely removed. In pathological cases, with pruning alone as above, the number of log files can be as high as \( N \cdot I \) with \( N \) groups and a checkpoint interval of \( I \) requests (proof deferred to [3]). So, GigaPaxos needs to infrequently (1) compact sparse log files, i.e., files with very few safety-critical entries; (2) merge them with other sparse log files; and (3) update the log index map entries in a consistent manner.

With all of the above mechanisms, GigaPaxos’ logger scales to a very large number of consensus groups while imposing negligible overhead when the working set fits in memory. Indeed, secondary storage, not memory, is what limits GigaPaxos’ group scalability. The worst-case disk storage overhead for \( N \) groups is \( O(INR) \), where \( I \) is the checkpoint interval and \( R \) the average request size, e.g., with \( I=100 \) and \( R=100B \), a machine needs over 1TB of storage to safely participate in \( N=100M \) groups.

### 3.6 Automated reconfiguration

A large number of consensus groups means it is impractical for an operator to manually reconfigure group membership, so GigaPaxos provides support for programmable policies that automate reconfiguration. For example, a principal could specify a simple policy to reconfigure upon, say, 10 requests from near an app-container location where it is not already replicated. Much more sophisticated policies including those optimizing global placement across all principals are straightforward to implement. We first describe the reconfiguration protocol below.

#### 3.6.1 Reconfiguration protocol

GigaPaxos’ reconfiguration protocol is similar to Liskov and Cowling’s Viewstamped Replication Revisited (VRR) [26], but differs in important ways. First, GigaPaxos uses an external reconfigurator (similar in spirit to Vertical Paxos[23]) that also integrates the function of group location, i.e., determining the current group for an object, a concern outside the scope of VRR (that suggests that clients could obtain this information from a “web site run by the administrator”). With a very large number of application RSMs and frequent reconfigurations, group location requires a systematic, scalable solution. Second, the reconfigurator for each application RSM itself must be replicated in order to prevent the application RSM from stalling permanently because of a reconfigurator failure.

Replicated GigaPaxos reconfigurators must agree on when to initiate a reconfiguration for an application RSM and on the composition of the new group as divergence can result in reconfigurators permanently losing track of the group. So each reconfigurator replica group is itself organized as an RSM whose state is the set of all application RSMs mapped to it via consistent hashing. In keeping with its completely event-driven design, programmatic reconfiguration in GigaPaxos is initiated by a client request (step 0) that happens to result in a demand report (step 1) from one or more app-containers to some reconfigurator(s). Upon receiving a demand report, any reconfigurator can propose an RC_INTENT(X) (step 2) command to reconfigure an application RSM X it manages and, when committed, the proposing reconfigurator in the common case single-handedly conducts the STOP/START/DROP reconfiguration sequence [26] for X (steps 3–7). When done, it proposes and commits RC_COMPLETE(X) (step 8) in its group. Persistently logging every state change in its RSM ensures that, upon the proposing reconfigurator’s failure or upon recovery, a reconfigurator can detect and complete unfinished reconfigurations of its managed application RSMs. A formal protocol description and a proof of correctness of the above reconfiguration protocol is deferred to a techreport [3]; that also describes how reconfigurators or app-containers themselves are added or removed (infrequently) by administrators.

#### 3.6.2 Extensible reconfiguration policy support

GigaPaxos enables applications to specify flexible policies that automate reconfiguration. Each reconfigurator RSM accepts periodic statistics about load or other metrics from any application RSM it manages and uses a customizable reconfiguration policy to decide whether and how to reconfigure the reconfiguree RSM. It is trivial also to let the application RSM simply send a request to its reconfigurator RSM when it deems a reconfiguration as necessary (or self-reconfigure as in VRR [26] and update the group location service), but allowing reconfigurators to make this decision allows implementing **global**
reconfiguration policies, i.e., policies that take into account statistics across many RSMs to make reconfiguration decisions for each RSM. Applications using GigaPaxos extend an abstract class, DemandProfile, to specify sophisticated reconfiguration policies based on failure, demand, or access patterns, performance, etc.

3.7 Safety and liveness properties

GigaPaxos without reconfiguration maintains safety and liveness properties identical to Paxos. With reconfiguration, liveness hinges on a majority of reconfigurators as well as app-containers being live and being able to communicate in a timely a manner (refer §7.4).

3.8 Replicable API and implementation

We implemented GigaPaxos with all of the features described above largely in Java with 23.9K semi-colons (83.6K newlines including documentation) of which 9.4K is for a stoppable Paxos implementation; 9.2K is for the reconfiguration protocol. The persistent logger uses an embedded database, Apache derby, by default, and also supports mysql. All transport is based on TCP; our nio library maintains and reuses a persistent connection to each machine, automatically attempts to create a new one if machine failures or other events cause I/O exceptions, and buffers a bounded number of messages to each destination to mask intermittent network failures. The size of an idle Paxos instance is ≈350B in our implementation because it is in Java; with a leaner language like C, it can be reduced further to ≈100B.

In order to remain agnostic to application-specific details, GigaPaxos requires an application to implement the following simple Replicable interface in order to be both replicable and reconfigurable, and an application may choose to use just one of the two features, for example, to create an unconfigurable RSM or reconfigure an unreplicated state machine (refer §7.3):

```java
boolean execute(Request request, boolean dontReply);
String checkpoint(String name);
boolean restore(String name, String state);
```

4 Evaluation

Our high-level goal is to quantify the costs and benefits of group scalability in GigaPaxos. We conduct the following experiments: (1) Comparison of GigaPaxos against state-of-the-art Paxos-based systems w.r.t. the number of supported groups and the impact on client-perceived performance; (2) Microbenchmarks evaluating the benefit and overhead of mechanisms in GigaPaxos; and (3) Case studies involving a number of third-party applications evaluating GigaPaxos’ usability and the benefits.

4.1 Group scalability comparison

We study the load vs. latency profile and the memory overhead for varying numbers of groups for three state-of-the-art systems that either are or comprise a consensus system, namely, ZooKeeper [18], OpenReplica [7], and Raft [31], compared to GigaPaxos. Unless otherwise specified, all experiments were performed on Amazon EC2 t2.medium (2 vCPUs, 4GB memory, and 8GB SSD disk) servers and sufficiently many c4.xlarge clients to saturate the servers in the same region.

4.1.1 Load vs. latency with varying no. groups

In this experiment, clients send requests at increasing rates to a single active RSM at servers that maintain varying numbers of mostly idle RSMs. There are 3 servers in all and each RSM’s consensus group is the set of all 3 servers. We measure the average response time over at least five runs each lasting 60s after discarding at least one or more warmup runs as needed to stabilize the servers. The request rate is increased until the system can not sustain that load, i.e., one or more servers either crashes, or the response time exceeds 1s, or the response rate drops below 99.9%.

Fig. 8 shows the load vs. response time profile of GigaPaxos, ZooKeeper, OpenReplica, and the Raft authors’ LogCabin [31] implementation for 1B no-op requests. Among the latter three systems, ZooKeeper scales to the highest capacity (32K/s) with a single group, but breaks down at barely hundred groups, while OpenReplica and Raft have significantly lower capacities with one group but don’t hit breakdown point until hundreds of groups. ZooKeeper scales to fewer groups in part because of the overhead of running separate JVMs with servers listening on different sets of three ports for each RSM, which, though cumbersome, we confirmed with its developer forums [6] as well as via code inspection was the most reasonable option to maintain separate consensus groups. Raft’s C++ implementation is leaner, so it scales to more
Figure 9: Group scalability: Load vs. latency for 1KB requests with varying number of idle groups.

All three systems show a stark, qualitatively similar degradation with increasing groups.

In contrast, GigaPaxos is fast, scaling up to 160K/s capacity with a negligible performance drop as the number of groups increases all the way to a million. Given that only a single group is active in this experiment, GigaPaxos mainly benefits from amortizing failure detection across groups, its holistic, single-process design, and its compact representation of idle Paxos instances.

Memory overhead. In order to measure the memory overhead of simply maintaining consensus instances (with zero request load), we pick two sets of server configurations, t1.micro EC2 servers with 1GB memory, and t1.small with 2GB memory. We then create as many consensus groups as the machine could sustain before hitting physical memory limits for each of the four systems. We compute the marginal memory overhead of consensus instances as 1GB divided by the difference between the number of groups respectively sustainable with 2GB and 1GB of physical memory.

Table 1 shows the marginal memory overhead of the four systems for idle consensus instances. GigaPaxos can create over 3 million Paxos instances per machine resulting for a cost of \(\approx 350B\) per instance (consistent with Fig. 4), which is orders of magnitude smaller than the other systems. We defer a more detailed breakdown of the constituent costs and the load vs. memory consumption behavior of the systems to the techreport [3].

Impact of request size. 1B requests measure the raw agreement throughput, but are hardly useful for any real application. We repeat the above experiment with 1KB requests and (because of space limits) show the results only for GigaPaxos and ZooKeeper in Fig. 9. Both systems are network bottlenecked\(^2\) and see a significant drop in capacity.

Table 1: Memory cost.

<table>
<thead>
<tr>
<th>System</th>
<th>Memory cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>GigaPaxos</td>
<td>346.4 Bytes</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>42.7 MB</td>
</tr>
<tr>
<td>OpenReplica</td>
<td>10.8 MB</td>
</tr>
<tr>
<td>Raft</td>
<td>5.4 MB</td>
</tr>
</tbody>
</table>

\(^2\)more precisely, by networking-related actions such as serialization and the TCP/IP stack, not the bare-metal bit/sec.

4.2 GigaPaxos microbenchmarks

4.2.1 Impact of batching on group scalability

The results above (§4.1) with a single active group may suggest that GigaPaxos is phenomenally group-scalable with no apparent costs, but that is hardly the case. Next, we stress-test GigaPaxos when a large number of groups are simultaneously active. We use 1KB requests and repeat the experiment above with the only difference that requests are sent in a round-robin manner across groups.

Fig. 10(a) shows that the throughput capacity of the system drops as the number of groups increases. The reason is reduced opportunities for batching requests. Opportunistic batching, i.e., without explicitly waiting for more requests to arrive, is well known to significantly improve the performance of Paxos-like protocols. However, it is in general not possible to batch requests across different RSMs as their group membership may be different. With increasing groups, GigaPaxos’ capacity drops until it hits \(\approx 22K/s\), which we have verified is its capacity with 1KB requests with batching disabled.

Fig. 10(b) shows an experiment similar to that in 10(a) but with 5- replica groups (instead of 3). Seemingly contrarily, the first set of bars show the capacity increasing with the number of groups. However, there is a simple explanation—coordinator load balancing—for this observation. As the number of groups increases from 1 to the total number of physical servers 5, the capacity increases because the coordinators for different 5-replica groups get randomly assigned to the servers. As the coordinator’s role—receiving every request and sending them as \texttt{ACCEPTs} to the group—is a key bottleneck, multiple groups naturally increase capacity. In contrast, the latter set of bars enable the \texttt{digest_requests} option wherein the entry replica broadcasts the request to all acceptors and the coordinator issues \texttt{ACCEPTs} only with request digests. Safety is preserved since an acceptor acknowledges an \texttt{ACCEPT} only if it has received the corresponding body.
Figure 11: (a) Group scalability “fine print” versus very large number of groups; (b) Fault scaling.

The benefit of coordinator balancing has been noted before, e.g., S-Paxos [10] proposes an optimization similar to GigaPaxos’ request digests (albeit with a more complex protocol), and others such as Mencius[28], E-Paxos[29] etc. [20] take different approaches to coordinator load balancing. In GigaPaxos, such optimizations are needed only when the number of groups is very small and the request size is not small ($\gg$ tens of bytes).

4.2.2 Hot swapping overhead

Table 11(a) summarizes the “fine print” limiting GigaPaxos’ group scalability. The experiments thus far considered up to a million groups that barely consume half a gigabyte of memory. However, 10 million instances is higher than what can be supported on the 4GB RAM servers. With such a large number of instances and a round-robin workload, every request encounters a paused instance, so the average latency is over 12ms compared to under 3ms for up to a few million instances (both measured under a round-robin light load of 100/s). Unpausing an instance currently requires two database lookups, one each for the paused instance state and the corresponding log index record (§3.5) that are currently paused and looked up independently; combining them (not yet implemented) will further reduce this penalty. The throughput takes a much more severe hit at 2.6K/s for 1KB requests vs. 22K/s for 1M groups (Fig. 10(a)).

4.2.3 Fault-tolerance

We defer two experiments: (1) fault-scaling, and (2) responsiveness to failures when subject to a node crash in a running system—to the appendix (§7.3.1).

4.3 Application usability case studies

In this section, we present several application case studies to show (1) that GigaPaxos’ Replicable API (§3.8) can be implemented easily for third-party applications, with or without intrinsic support for replication, so as to make them replicable and reconfigurable; (2) the latency benefit of object-group configurability.

4.3.1 myCloud: Share document editing and storage

We implemented a Replicable wrapper for etherpad [1], an open-source, document editor that allows users to collaboratively edit documents or “pads” in real-time via a web browser (similar to the popular, proprietary Google Docs). etherpad does not intrinsically support replication or fault-tolerance. Client libraries for its API are available in a variety of languages; we used the Java API [2] to make it fault-tolerant and reconfigurable via the Replicable wrapper. We also used GigaPaxos’ general-purpose support for applications to easily delegate messaging of replies back to the originating client.

In this wide-area experiment, we deploy 7 etherpad servers respectively at California, Frankfurt, Ireland, Sydney, Seoul, Tokyo, and Virginia. A GigaPaxos client creates a single pad using the createService() client API, which by default is set to create an RSM group of all 7 replicas. A controller script in our lab then emulates a “mobile” etherpad client that trots across different cities as shown in Fig. 12 sending tens of requests from each city. The DemandProfile policy is designed to reconfigure the RSM once every 20 requests to either 1 (blue/solid) or 3 (green/dashed) closest app-container locations.

Fig. 12 shows that the carefully-chosen 3-replica RSMs can significantly reduce end-to-end client-perceived latency, sometimes by over 200ms. The 1-replica RSM as expected yields the lowest latency but only ensures durability, not availability amidst failures, and is included just to show the best-case. Reconfiguration itself roughly takes as much time as 2-3 Paxos operations, so requests are occasionally lost when sent to an app-container where the Paxos instance no longer exists.

4.3.2 Usability and performance overhead

Implementing Replicable for etherpad was rather easy and involved just 60 lines of code to GigaPaxos’ abstract, general-purpose “hello world” client and application classes to support etherpad’s three basic request types used in this experiment; supporting its full API will increase the integration work.

We have implemented Replicable wrappers for a number of third-party applications as listed in Table 2 including OpenKM [4] (comparable to Google Drive) and popular key-value stores. Despite the simplicity, some of the wrappers are powerful, e.g., the mysql wrapper
is schema-agnostic and anyone can reuse it to designate either each row or each table as an independently reconfigurable RSM. We could do this because Replicable's checkpoint and restore methods can naturally avail of sqldump to checkpoint a single record, table, or database in a schema-agnostic manner. A detailed study showing that GigaPaxos adds only a small penalty to the single-server capacity of these applications is deferred [3].

<table>
<thead>
<tr>
<th>Application</th>
<th>#semi-colons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etherpad</td>
<td>60</td>
</tr>
<tr>
<td>OpenKM [4]</td>
<td>89</td>
</tr>
<tr>
<td>MySQL</td>
<td>79</td>
</tr>
<tr>
<td>Cassandra</td>
<td>78</td>
</tr>
<tr>
<td>Mongo</td>
<td>53</td>
</tr>
<tr>
<td>Redis</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 2: LOC for Replicable wrappers.

With the help of tutorials and starter code, GigaPaxos has been in used by a small distributed systems class of 11 students (6 undergraduate, 5 masters) to implement a simple, in-memory map that is gratuitously durable, fault-tolerant, per-key reconfigurable, and ultra-fast (Fig. 8(a)) for small keys and values; and it is being used again in a 30+ student class for the second time.

5 Limitations and open questions

How much group scalability is enough? That question has not been directly studied in this paper and it is not easy to quantitatively answer it. On a single machine, accommodating a large number of groups increases performance primarily because it increases concurrency. Many applications commonly (e.g., mysql, Cassandra, etc.) see improvements of $>10 \times$ (e.g., refer §7.5) with hundreds of threads per machine for mixed read/write workloads compared to using a single thread. However, common-case throughput is not the only metric of interest. For example, if edge services like mysql become commonplace, a large outdoor event (e.g., Times Square on New Year’s eve) may well involve a million people connected via the same virtualized LTE base station. A system not designed for group scalability cannot easily be used to quickly regroup a million sets of objects and create a replica of that group at that virtualized base station. The practical alternative of using a few dozen pre-configured replica groups (e.g., east coast, west coast, etc.) is good enough for today’s modestly geo-distributed clouds. So, is our envisioned scenario of massively geo-distributed micro-clouds on virtualized base stations misguided and unnecessary? Or is it being held back by the lack of system support for massive-scale service dispersability? That is a question our work might help answer.

State-of-the-art alternatives. Although much of the performance microbenchmarks presented here compared against ZooKeeper, the two are not directly comparable. GigaPaxos is designed to allow arbitrary black-box applications to be easily encapsulated as a reconfigurable RSM, but ZooKeeper does not provide a general-purpose RSM abstraction but only a fault-tolerant object store that is not designed for agile reconfigurability. The closest object-store alternatives to GigaPaxos are systems like Spanner or Auspice, but neither provides a general-purpose RSM abstraction for third-party blackbox applications with high group scalability.

Multi-group transactions. GigaPaxos is designed for use cases where the principals are completely independent, i.e., it provides per-principal linearizability (or weaker semantics) but does not offer any ordering guarantees across principals. Our experience with GigaPaxos suggests that support for multi-group transactions is useful in many scenarios, and that design and implementation is ongoing work. Our preliminary work suggests that a generalization of two-phase commit to RSM groups ([17, 13]) at a high-level suffices to retain per-group linearizability with multi-group transaction support.

Isolation for security. GigaPaxos’ design assumes that all of the service principals mutually trust each other; in particular, in our implementation, all requests at a GigaPaxos server node are processed within a single JVM. The design and implementation of mechanisms for using GigaPaxos with mutually untrusted service principals requires more isolation with commensurate performance overhead. Using GigaPaxos on untrusted third-party hardware further requires byzantine fault-tolerant consensus with even higher overhead. GigaPaxos currently does not support such mechanisms, but they will be needed for the massively geo-distributed edge cloud scenarios we envision, and need more research.

6 Conclusions

Perhaps because of pedagogical challenges or because of the “costly” mental image replication already invokes, consensus implementations today inherently embed assumptions appropriate for monolithic applications. We presented GigaPaxos, a novel system that enables group-scalable replicated state machines, easily allowing any application to create an object managed by a consensus group on the fly and reconfigure it as needed. We have conducted a number of application case studies to show that agile reconfiguration and reconfigurability in GigaPaxos can significantly improve client-perceived performance. GigaPaxos is a synthesis contribution combining simple ideas and meticulous engineering into a package that is much more than the sum of its parts. The GigaPaxos code with tutorials, case study example code, and documentation is available at: http://anonymizedforpeerreview.org

References


[5] Personal communication with members of the Google Spanner team.


7 Appendix

7.1 Paxos and RSM primer

Paxos (or Multi-Paxos) is a consensus algorithm that enables a set of nodes to agree on a growing sequence of values over time despite node failures and unbounded network delays. As shown in Fig. 13, Paxos consists of two phases: the first allows one (or more) node(s) to assume the role of a coordinator, and the second determines a decision value for each position in the sequence. With long-lived coordinators, only the second phase is needed for each new value in the common case. Paxos guarantees the agreement safety property that two different decision values for the same slot are never delivered to any (including singleton) subset of nodes. Progress, i.e., any values being agreed upon at all, is ensured only when a majority of nodes are up and can communicate with each other in a timely manner.

In the replicated state machine (RSM) approach, the values are client requests and nodes in a consensus group supply the sequence of requests to their locally running replicas of an arbitrary (but identical) deterministic application. A common optimization is to batch multiple client requests within each value, which helps when agreement throughput as opposed to application execution [35] is the bottleneck.

7.2 Group creation: phantoms and corpses

With a large number of Paxos groups, some groups may get created when a member machine has failed. Upon recovery, \texttt{PaxosManager} on the machine will have no checkpointed or paused state for this instance. It is impractical for a recovering machine to contact all other machines for a list of all recently created Paxos groups, and for alive group members or reconfigurators to keep polling the failed member machine after group creation without introducing more instance-specific state and messaging overhead. For example, with 100 million Paxos groups, 100 machines, and an average group size of 5, each machine will have on average 5 million Paxos instances, so any additional instance-specific state and messaging unless ephemeral can significantly increase memory and bandwidth consumption.

Instead, GigaPaxos machines adopt a lazy approach to create Paxos groups whose birthing they missed. When a machine receives a protocol message for a phantom Paxos instance, i.e., for which it has no state, it assumes that the group may have gotten created in its absence, and contacts the sender of the message to enquire about the group’s membership so as to create the corresponding Paxos instance locally. Care needs to be taken to only create Paxos instances, identified by the \texttt{(name,epoch)} two-tuple, for the same name with strictly increasing epoch numbers for safety. The reconfiguration protocol ensures this invariant by preserving the most recent epoch’s final checkpoint, taken synchronously and immediately after executing corresponding \texttt{STOP} request, for a long threshold expiry time. Even though it would be futile to try to recreate a Paxos instance right after it was explicitly deleted, say, because of the receipt of a protocol message from a laggard member, it is unnecessary overhead to check the disk upon receiving phantom protocol messages; instead, the manager keeps deleted Paxos instance “corpses” in an in-memory “morgue” map for a fixed timeout, and tries to create a new Paxos instance if and only if a matching corpse is not found in...
the morgue.

7.3 Replicable application and client API

In order to remain agnostic to application-specific details, GigaPaxos requires an application to implement the following simple Replicable interface in order to be both replicable and reconfigurable.

```java
boolean execute(Request request, boolean dontReply);
String checkpoint(String name);
boolean restore(String name, String state);
```

The flag `dontReply` is useful during (1) recovery to inform the application to withhold interaction with the end-client while rolling forward, and (2) regular execution to hint that only the “entry” replica need reply back to the end-client. The interface assumes that an application `Request` is serializable to a string and returns the RSM name via a `getServiceName()` method. The GigaPaxos logger maintains checkpoints up to a configurable maximum size in its own database (distinct from the application’s if any); for large checkpoints, the application can also use the string `state` as simply an application-specific handle, e.g., a file name or URL, a mechanism also internally used by GigaPaxos’ reconfigurators whose state can be very large.

The client API simply consists of `createService(X, S)` (deleteService(X)) to create (delete) a service X with initial state S, and `sendRequest(R)` to send a Request R to the RSM; querying reconfigurators to locate and select the closest app-container replica is internally handled. Upon creation, reconfigurators randomly choose the initial set of app-containers that later get reconfigured automatically.

7.3.1 Fault scaling

Fig. 11(b) shows the impact of the size of the group on the capacity with 1KB requests and a single group. As expected, the capacity decreases as the replication factor increases because of the increase in message overhead. With multiple groups (not shown), this trend w.r.t. the replication factor remains unchanged; for any given replication factor, the aggregate throughput initially remains steady as the number of groups increases up to the replication factor because of coordinator load balancing (Fig. 10(a)), but then decreases with more groups because of reduced batching up until it matches the unbatched capacity.

7.3.2 Failure recovery

Can GigaPaxos groups recover quickly from failures? To study this, we use 5 servers either with one group, as in Fig. 14(a), or with 5 and 1000 groups respectively as in Fig. 14(b). In the 1-group case, the 6 numbered marks respectively correspond to the (1) failure of an acceptor; (2) recovery of the acceptor; (3) failure of two acceptors; (4) recovery of the two acceptors; (5) failure of the coordinator; (6) recovery of the coordinator.

We find that the throughput initially decreases but eventually increases upon the failure of acceptors and is chugging along most happily after the failure of two acceptors (mark 3) as the coordinator has to do less messaging. This increase in throughput with acceptor failures is different from the finding in the ZooKeeper paper [18] (that inspired this experiment) where the throughput roughly decreases by the share of requests going to the failed nodes. Unlike Zab [19] that pins each client to a single server (to preserve primary order causal consistency), GigaPaxos clients can send requests to any server.

But coordinator failures do force a downtime commensurate to the failure detection timeout, which defaults to 6s in our implementation and is automatically increased with more machines if the aggregate probe rate at a machine exceeds 10/s.

The experiment in Fig. 14(b) is similar to 14(a) with the only difference that the labels “acceptor” and “coordinator” are not meaningful here because different groups’ coordinators get randomly mapped on to different servers. As a result, the aggregate throughput never goes down to zero as there are always some groups whose coordinators are alive. The 1000-group case does plunge significantly upon a machine failure because, in addition to the 0–6s of downtime for roughly a fifth of the groups, there is also the added message overhead of electing new coordinators for those groups. More generally, in a GigaPaxos system with M machines, a total number N of consensus groups, and a total number n of active groups, the failure of a single machine will in expectation temporarily stall N/M groups and induce n/M elections (as paused groups do not elect a coordinator until stirred by a client request).

7.4 Safety and liveness properties

GigaPaxos preserves safety despite arbitrary failure patterns for each Paxos group by maintaining the critical Paxos invariant as-is, namely, if a coordinator c issues a proposal for request r for slot n in its ballot (b, c), then there exists a majority of acceptors such that either none of them have accepted any proposal for slot n in ballots lower than (b, c), or r is the request accepted for slot n in the highest ballot less than (b, c) by any acceptor.
in that majority\textsuperscript{3}, GigaPaxos does not provide any ordering, consistency, or isolation properties for operations across different RSMs currently. Hot swapping preserves safety as the set of actions performed by a a GigaPaxos instance is a subset of actions that an acceptor or coordinator could have performed even without hot swapping.

GigaPaxos ensures liveness for each RSM during periods of synchrony when a majority of acceptors are available. Hot swapping only has a performance impact when the number of consensus groups is very large, and a significant fraction of GigaPaxos machines across all groups have failed. Persistent logging at acceptors ensures that, even if all acceptors crash at some point, the group eventually makes progress during subsequent periods of synchrony and majority availability. Without this, manual intervention would be required to safely recover from the crash of a majority of group members, which is impractical for a large number of groups.

7.5 Concurrency gains of group scalability

To show that the overall throughput of a replicated database application can improve significantly with repliconfigurability and group scalability, we conduct a simple experiment using the OLTP benchmark. We first test the transaction throughput of a MySQL database for the following scenarios: (1) consider the whole database as a single principal, and send transactional requests back-to-back to the database, (2) consider each table in the database as a principal and send transactional requests back-to-back to each table in the database. Here we assume the tables are independent with other, i.e., the transactions to update different tables can run in parallel. We use the MySQL connector/J JDBC library to measure the throughput capacity. The total time for each round of a single experiment is 30 seconds. The read/write rate is 0.1, i.e., 10% read with 90% write operations. The size of each record is 200B. We keep increasing the number of tables in the database and show the overall throughput in Figure 15. It shows that the ratio between the second and first scenario is $18 \times$, i.e., the overall throughput is increased by $18 \times$ with each table being taken as a single object compared to the case that the whole database is taken as a single object.

Since GigaPaxos itself consumes CPU, I/O and network network resources, $18 \times$ is only an upper bound on the achievable concurrency benefits. Next, we show the ratio after integrating GigaPaxos with the MySQL application used in the last experiment. We wrap MySQL with GigaPaxos' Replicable API, and measure the overall throughput of this replicated database system for transactions to different tables. We compare two cases: (1) all the tables are in a same fault-tolerance group, (2) each table is in a single fault-tolerance group. We use three replicas (EC2 t2.medium) as our hosts. Each host runs both GigaPaxos and MySQL. We use a single client to send requests to different groups in a round-robin manner as the single client is enough to saturate the replicated database system. The result is shown in the Figure 15. As it shows, the overall throughput is improved by 2x with different tables in different Paxos groups.

\textsuperscript{3}This invariant is usually stated without referring to the slot number $n$ \cite{22} as a coordinator can use its ballot to issue any number of different proposals each with a different slot.

![Figure 15: Throughput improvement: multiple groups v.s. single group](image-url)