GeoPaxos+: Practical Geographical State Machine Replication

Paulo Coelho
Federal University of Uberlândia, Brazil

Fernando Pedone
Università della Svizzera italiana (USI), Switzerland

***Abstract*—In some online services, the geographical location of a client tends to determine the data accessed by the client's requests. Geographical locality holds, for example, in location-based services, tracking systems, and social networking services. State machine replication protocols can use geographical locality to optimize performance by ordering requests efficiently. In order to be effective, though, two requirements must be fulfilled. First, protocols must identify the data accessed by a request before the request is executed. Second, protocols must determine which parts of the service state are accessed where and with what probability. The paper presents a geographical state machine replication protocol that meets both requirements. We illustrate the use of our protocol by developing a geographically replicated B+Tree service. We fully implemented the B+Tree service and show experimentally that it outperforms implementations based on classic (i.e., Paxos) and recent (i.e., EPaxos) general-purpose replication protocols by a large margin.**

***Index Terms*—fault tolerance, state machine replication, geo-replication.**

I. Introduction

Geographical replication consists in distributing replicas of a service across different geographical locations. It is typically used to improve performance (e.g., clients can access a nearby replica) and to tolerate catastrophic failures (e.g., a flooding that renders a datacenter inoperative may not affect a remote datacenter) [1], [2]. Ensuring that geographically distributed replicas behave consistently, however, is challenging. This is because strong consistency (e.g., linearizability) requires replicas to coordinate over wide-area links, which introduces inherent communication delays.

In order to provide strong consistency in the presence of wide-area delays, system designers have come up with approaches that exploit specific characteristics of the service. One class of protocols uses application semantics to reduce the number of communication steps needed to order client requests before they can be executed by the replicas (e.g., EPaxos [3], M2Paxos [4], MDCC [5]). Another class of protocols takes advantage of client access patterns present in some online services (e.g., [6], [7]). In services that exhibit *geographical locality*, the geographical location of a client tends to determine the data accessed by the client's requests. Geographical locality holds, for example, in location-based services, tracking systems, and social networking services [8].

In state machine replication [9], [10], client requests are first serialized and then executed by the replicas. The serialization of requests usually relies on one replica, the *serializer* (i.e., leader, coordinator) [11], [12], which proposes the order of requests to the other replicas. If the serializer fails, a backup replica takes over. In geographical settings, clients near the serializer have a performance advantage since their requests are ordered more quickly. GeoPaxos [6] improves client performance with multiple geographically distributed serializers, each serializer responsible for ordering requests submitted by nearby clients. To ensure consistency, requests that access a common part of the state are ordered by the same serializer (or its backup, if the serializer fails). More concretely, the service state, hereafter assumed to be composed of objects, is logically partitioned so that each partition is under the responsibility of a serializer. A request is ordered by serializer S if the request accesses objects in S's partition. Since an ideal partitioning of the service state is unlikely, some requests may involve multiple serializers. In this case, the serializers involved in the request must coordinate to order the request. For example, if X and Y are objects in the service state, then replicas S_X and S_Y would be responsible for ordering requests that access only X and only Y, respectively. Requests that access X and Y require S_X and S_Y to coordinate. In the presence of geographical locality, S_X (S_Y) would be a replica near the clients likely to access X (Y).

GeoPaxos relies on two important assumptions. First, on a selection of serializers that optimize for client access patterns. This is difficult because access patterns may not be known in advance and they may change during the execution (e.g., since clients change their location). Besides, some objects may be accessed by clients in multiple locations, and thus, there is no optimal serializer. Second, replicas must identify the objects accessed by a request before the request is ordered. This is necessary because a replica must know whether it should serialize the request and, if so, whether it must coordinate with other serializers. Determining the objects accessed in a request is easy in simple services (e.g., a key-value store, where a request either reads or writes a key-value object) but difficult in arbitrary services (e.g., dynamic data structures, where a request accesses objects depending on the value of other accessed objects).

In this paper, we present GeoPaxos+, a geographical state machine replication protocol that addresses the issues raised above. GeoPaxos+ extends GeoPaxos [6] as follows. First, GeoPaxos+ allows objects to have multiple serializers. This has the advantage that clients in different geographical locations can efficiently read an object by using a nearby serializer, at the cost of more expensive update requests, which must involve all the serializers of the object. Second, we introduce a model to optimize the performance of geographically distributed applications. Our model determines which replicas are best suited to act as serializers. The model accounts for multiple input parameters including the location of clients, wide-area delays, and client access patterns. Third, we propose heuristics to speed up the optimization process. Fourth, we introduce a B+Tree service to show how one can cope with the challenges involved in replicating a complex service. A B+Tree is a dynamic data structure with interesting access patterns: parts of the tree are mostly read by clients, notably the root, while updates happen mostly near the leaves of the tree. Moreover, some variations of a B+Tree have been used in location-based services [13]. We experimentally evaluate our techniques and show that they outperform classic state machine replication (i.e., Paxos) and recent semantic-based approaches (i.e., EPaxos).

The remainder of the paper is structured as follows. Section II describes the system model and the background. Section III introduces the proposed optimization model. Section IV illustrates how to cope with complex services with a B+Tree. Section V describes our prototype, while Section VI discusses the experimental evaluation. Section VII surveys related work and Section VIII concludes the paper.

II. Background

In this section, we define our system model (§II-A) and recall the basics of GeoPaxos (§II-B), the underlying geographical replication protocol we build upon.

A. System model and consistency criterion

We consider a message-passing distributed system: *client* and *server* processes (i.e., *replicas*) are geographically distributed and communicate using FIFO channels. Each replica stores the complete application state (i.e., full replication). Processes are subject to crash failures and do not behave maliciously (e.g., no Byzantine failures). The system is asynchronous (i.e., no bound on message delays and on relative process speeds), augmented with additional assumptions to implement Paxos [14], [12] (i.e., leader election).

The consistency criterion is *linearizability* [15]. An execution is linearizable if there is a permutation of the client requests in the execution that respects (i) the service's sequential specification and (ii) the real-time precedence of requests as seen by the clients. Request op_i precedes request op_j if the response of op_i occurs before the invocation of op_j.

B. Geographical replication with GeoPaxos

We now briefly describe GeoPaxos (for more details, including a discussion about GeoPaxos's correctness, see [16]). A client submits a request to a nearby replica, which propagates the request to the other replicas to be ordered before the request is executed by each replica. The main insight of GeoPaxos is to allow different replicas (*serializers*) to order requests, depending on the objects accessed in the request. The system assumes a mapping of objects to serializers, known by all the replicas. If object X is mapped to replica A, we say that A is a serializer for X, and any requests that access X must be serialized by A. When a replica first receives a request, it determines all the objects accessed in the request. If the replica is the serializer for an object accessed in the request, it computes a unique timestamp for the request and propagates the timestamp to all replicas. If the request has a single serializer (i.e., all the objects accessed by the request have the same serializer), then the proposed timestamp is the request's final timestamp, used to order the request. If the request has multiple serializers, then timestamps from all serializers involved are needed to determine the final timestamp of the request, computed as the maximum among the proposed timestamps.

Fig. 1 illustrates the execution of single- and multi-serializer requests. The contacted replica determines the objects accessed in the request and propagates the request to the other replicas. Upon receiving a request, a replica waits for the request timestamp. If the replica is a serializer of the request, it computes a final timestamp (if the request is single-serializer) or a tentative timestamp (if the request is multi-serializer) and sends it to all replicas. Once the final timestamp of the request is known by a replica, the request is enqueued for execution. Request op is executed once it has a final timestamp and there is no ongoing request op' with a final or tentative timestamp smaller than op's timestamp. This is needed to ensure that requests are executed in timestamp order.

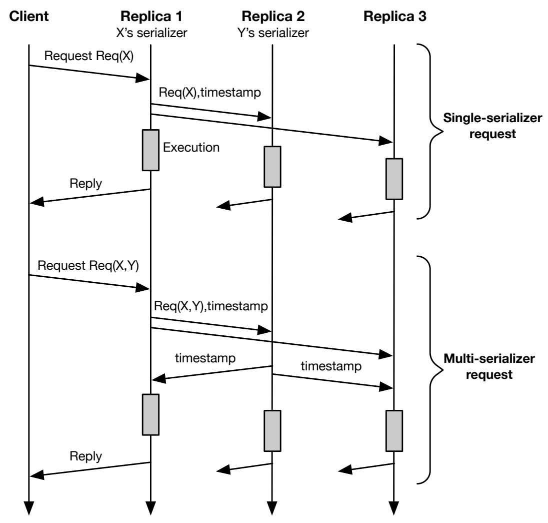


Fig. 1: GeoPaxos execution modes.

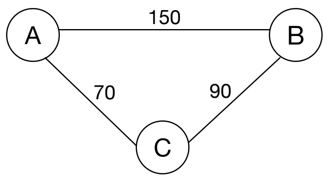
Serialization assigns unique timestamps to requests using Lamport timestamps. To tolerate failures, a serializer replicates this state using Paxos (for clarity, we omitted serializer replicas in Fig. 1). A serializer replica only responds to the serializer, as part of Paxos, after it has received the corresponding request. Thus, when a replica executes a request, the request has been replicated in enough replicas.

III. Optimizing object access

In this section, we extend GeoPaxos to allow objects with multiple serializers (§III-A), introduce our optimization model (§III-B), illustrate its use (§III-C), and present heuristics to speed up optimization in geographical settings (§III-D).

A. *From single to multiple serializers per object*

The choice of a serializer is key to performance. Consider, for example, the deployment shown in Fig. 2(a), where A, B and C are the serializers for objects X, Y, and Z, respectively. Graph edges show the latency between replicas in milliseconds. Now assume a client sends a request on object Z to B. Even though every replica has a copy of all objects, B should wait for C to order the request before the execution. In this situation, the client would wait for a full round-trip between replicas C and B (plus the communication needed by the replication of C's serialization state) before receiving a reply. If object Z is mostly accessed through replica B, then it would be more efficient to assign B as Z's serializer. However, since Z is also frequently accessed by clients near C (see Fig. 2(b)), neither B nor C are ideal to act as serializer of requests on Z.



(a) Topology graph: edges represent latency in milliseconds.

Workload for $\boxed{X}$

Replica	A	B	C
Reads	10	100	50
Writes	10	1	5

(b) Workload for object X.

Fig. 2: Topology and workload for three replicas in a WAN (A is X's serializer, B is Y's serializer, and C is Z's serializer).

We now revisit GeoPaxos and introduce multiple serializers per object. In order to allow an object to have *multiple serializers*, we distinguish between operations that only read the object from operations that read and update the object. A request that only issues read operations on an object can be ordered by any serializer of the object, while a request that issues a write operation on the object must be ordered by all the serializers of the object. This is inspired by the read-one write-all (ROWA) replication approach, a particular case of general quorum-based systems [17].

In the example of Fig. 2, we could assign object Z serializers B and C. Thus, clients that connect to B or C can have requests that read Z ordered quickly, while requests that modify Z must be ordered by both B and C. Finally, a request that reads X and writes Z will be ordered by serializers A (because of the read operation on X), and B and C (because of the write operation on Z).

B. *Optimization model*

In order to determine the serializers of an object, we need to monitor the frequency of read and write accesses to the object, that is, the *workload* of the object. Moreover, we need

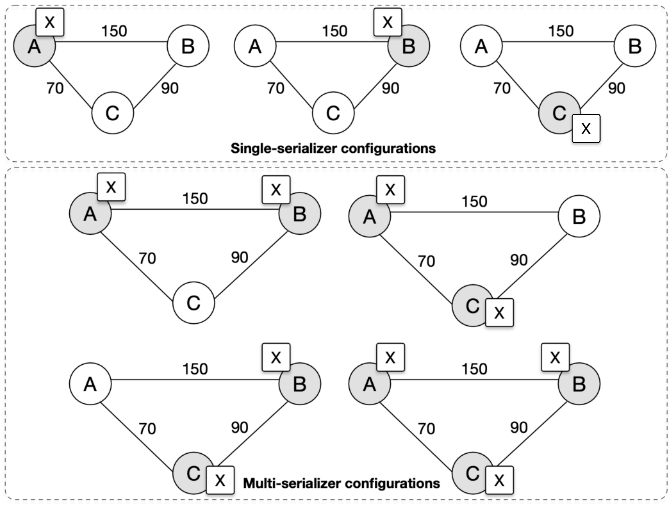

X
A
150
B
70
90
C
Single-serializer configurations
Multi-serializer configurations

Fig. 3: Configurations for object X ($\boxed{X}$: serializer for X).

information about the system *topology*, that is, the approximate communication latency between replicas. The topology is a graph that represents the cost between pairs of replicas in terms of latency. Any other metric that reflects meaningful costs between replicas may also be used (e.g., link bandwidth). We define $\delta_{A,B}$ as the latency between replicas A and B, represented by the edge weight in Fig. 2(a) (150 in this case). Workload is a per-object information and represents the number of read and write operations that clients have issued in some pre-defined time frame. In Fig. 2(b) clients connected to A read object X 10 times, while clients connected to C write 5 times to X.

With the topology and the workload, we assign one or more serializers to each object in such a way that it minimizes the final cost and the overall operation latency. The intuition is that in a read-intensive workload, multiple replicas can order read operations for a specific object independently, i.e., read operations are local to each replica (low latency) and scale with the number of replicas. However, if there are many write operations on an object from a specific replica, this replica should be the serializer for this object to avoid expensive multi-replica write operations.

We now derive a general optimization model to compute the serializers of an object. Fig. 3 exhibits the possible configurations $\mathcal{N}$ for object X in the system described in Fig. 2(a). A configuration $\mathcal{P}_{X,i}$ is a set of serializers for X. The total number of possible configurations is $\sum_{i=1}^{N} \binom{N}{i}$, where N is the number of replicas in the system. Each configuration $\mathcal{P}_{X,i}$ for object X has cost $\mathcal{C}_{X,i}$, which depends on the number of read and write operations on X and the latency to perform reads and writes in $\mathcal{P}_{X,i}$:

$$\mathcal{C}_{X,i} = \sum_{g=1}^{N} w_{X,g}(W_i + \delta_g) + r_{X,g}(R_i + \delta_g) \qquad (1)$$

In equation (1), $w_{X,g}$ and $r_{X,g}$ are the number of reads and writes on X issued at replica g. R_i and W_i are the latency of a single read or write operation involving serializers in $\mathcal{P}_{X,i}$. Since we can always read from a serializer, $R_i = 0$; since a write operation must involve all serializers in $\mathcal{P}_{X,i}$, $W_i = max\{\delta_{k,h} \mid k, h \in \mathcal{P}_{X,i}\}$. If g is not a serializer in $\mathcal{P}_{X,i}$, then both read and write operations will have an additional cost δ_g to reach the closest serializer to g, computed as $\delta_g = min\{\delta_{g,h} \mid \forall h \in \mathcal{P}_{X,i}\}$.

The ideal configuration, $\mathcal{P}_{X,best}$, has the smallest cost, as defined in Eq. 2:

$$\mathcal{C}_{X,best} = min\{\mathcal{C}_{X,i}, \forall \mathcal{P}_{X,i} \in \mathcal{N}\} \quad (2)$$

C. Example

To illustrate our optimization model, consider the deployment shown in Fig. 2. In such a case, we have $N = 3$ and $\binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 7$ possible configurations. Table I details the cost for each configuration in our example. We can observe that cost $\mathcal{C}_{X,best} = 2400$ corresponds to the configuration with object X serialized by replicas A, B and C. In this case, the cost of a write operation, W_i, is given by $\delta_{A,B} = 150$, since this is the maximum latency between replicas. For replicas A, B and C, we have $\delta_{A,best} = \delta_{B,best} = \delta_{C,best} = 0$, since A, B, and C are serializers for object X in this configuration. Most operations are reads from clients connected to replicas B and C, which can be served locally by a replica; more than half the number of writes are from clients connected to replica A (see Fig. 2(b)).

$\mathcal{P}_{X,i}$	W_i	R_i	$\delta_{A,i}$	$\delta_{B,i}$	$\delta_{C,i}$	$\mathcal{C}_{X,i}$
$\{A\}$	0	0	0	150	70	19000
$\{B\}$	0	0	150	0	90	7950
$\{C\}$	0	0	70	90	0	10490
$\{A,B\}$	150	0	0	0	70	6250
$\{A,C\}$	70	0	0	90	0	10210
$\{B,C\}$	90	0	70	0	0	2840
$\boldsymbol{\{A,B,C\}}$	150	0	0	0	0	**2400**

TABLE I: Cost $\mathcal{C}_{X,i}$ per configuration $\mathcal{P}_{X,i}$ for object X.

D. Speeding up optimization

The complexity of our optimization function grows linearly with the number of objects and exponentially with the number of replicas. This section presents strategies to speed up the execution of our optimization model.

1) Hot replicas: The approach presented in §III recalculates preferred replicas for all tree nodes that have changed since the last computation. The *hot replicas* technique reduces the number of combinations that we have to consider for each node: whenever the number of operations issued by clients to a replica g is within a configurable threshold of the other replica's operations count, configurations where g is serializer are not considered. In the workload in Fig. 2(b), since replica A receives fewer operations than any other replicas, configurations involving A could be skipped in the calculation, reducing possible configurations from 7 to 3. The threshold for skipping a replica represents a compromise between faster calculation and accuracy.

2) Least Recently Used (LRU) Caching: The idea is to leverage two important aspects that can be observed in GeoPaxos+. The first observation is that, given two similar "enough" workloads, serializer assignments will be the same, which means that we can cache computed results. The second observation is that changes in the workload typically happen gradually over time, meaning that there is a certain probability that recent calculations can be re-used.

Previous calculations are cached in a map data structure providing retrievals in constant time. In such a map, keys are a combination of normalized workload and update ratio, which are the two aspects the calculations depend on. Intuitively, the normalized workload captures the proportion of operations received from different replicas, while the update ratio represents the proportion of read and write operations for each replica. We limit the size of the cache using an LRU strategy.

The normalized workload for object X at replica g, $\mathcal{L}_{X,g}$, is the amount of total operations on X at g compared to the total number of operations on X at all replicas, and can be expressed as:

$$\mathcal{L}_{X,g} = \frac{w_{X,g} + r_{X,g}}{\sum_{h=1}^{N}(w_{X,h} + r_{X,h})} \quad (3)$$

The update ratio, $\mathcal{U}_{X,g}$, is defined as the relation between writes and total number of operations for g:

$$\mathcal{U}_{X,g} = \frac{w_{X,g}}{r_{X,g} + w_{X,g}} \quad (4)$$

Considering the example in Table 2(b), we have normalized workload $\mathcal{L}_A = (20)/(176) \approx 0.11$ and update ratio $\mathcal{U}_A = 10/20 \approx 0.50$. We cap the results at two decimal digits to reduce the number of combinations, increase the hit ratio, and use the result as part of the key to look up an entry in the map data structure. The final key is the concatenation of the normalized workload and the update ratio without the decimal point, in this case 011050.

IV. Distributed B+Tree

In this section, we present an overview of our B+Tree implemented with GeoPaxos+ (§IV-A), discuss the B+Tree algorithm in detail (§IV-B), and explain how the optimization model is recomputed on the fly (§IV-C).

A. Overview

A B-tree is composed of a root node, internal nodes, and leaves with a variable but often large number of children per node. The root may be either a leaf or a node with two or more children. A B+Tree is a B-tree in which each node contains only keys (not key-value pairs) with an additional level at the bottom with linked leaves. A typical use of a B+Tree is storing data for efficient retrieval in a block-oriented storage context, like file systems, primarily because a B+Tree has a very high fanout (i.e., the number of pointers to child nodes in a node).

Fig. 4 shows a B+Tree as implemented by GeoPaxos+ (with fanout equal to three). Each tree node is an object containing fields to store serializer information (set G in Fig. 4), and counters for monitoring reads and writes from each replica (see Table II). For example, the root of the tree has replicas A and B as serializers. Therefore, reading the root must involve either serializer A or B; updating the root must involve both A and B.

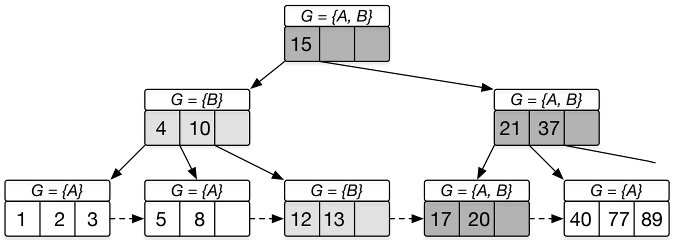


Fig. 4: B+Tree with serializers G for each node.

Field	Type	Description
uid	`long`	unique identifier
depth	`int`	depth in the tree
parent	`pointer`	parent node (∅ if root)
is_leaf	`boolean`	whether it is a leaf or not
size	`int`	current number of keys
keys	`key_type[]`	sorted array of keys
values	`pointer[]`	pointers to the next node/value
reps	`int[]`	serializers (set G in Fig. 4)
changed	`boolean`	whether it has changed
reads	`int[]`	number of reads from each replica
writes	`int[]`	number of writes from each replica

TABLE II: Structure of a distributed B+Tree node (in gray the data used by GeoPaxos+).

Clients send requests to the closest replica in the form of a tuple $(\textsc{op}, k, v)$, where OP is one of the requests listed in Table III, k represents the key on which the request will be executed, and v is the value (or $\perp$, when no value is required). A request to read key 13 in the tree can be serialized by replica B alone, since B is one of the serializers of the root node, the left child of the root, and the leaf node containing 13.

Operation	Description
READ$(k, \perp)$	returns value of key k
GET-NEXT$(k, \perp)$	returns value of the smallest key bigger than k
GET-PREV$(k, \perp)$	returns value of the biggest key smaller than k
UPDATE(k, v)	updates the value of key k and returns the old one
INSERT(k, v)	inserts key k and value v

TABLE III: Operations on a B+Tree node.

Since GeoPaxos+ periodically recomputes serializers based on the frequency of read and write operations on tree nodes, it may happen that after a replica computes the serializers of a request and the request is ordered by these serializers, the mapping of objects to serializers has changed. We explain next how GeoPaxos+ detects and handles such cases.

B. Detailed algorithm

When a replica r receives a request from a client, it computes the serializers involved in the request using function *psite* in Algorithm 1 (line 17). Function $find$ has two input parameters, the local tree $\mathcal{T}$ at the replica and the key $op.k$ accessed in the request; it returns the leaf node in $\mathcal{T}$ that contains (or would contain) $op.k$.

The serializers involved in the request depend on the type of the request (see Table III). A READ operation (line 24) is always serialized by a single replica, either r, if r is a serializer of the object read, or the closest serializer g to r (i.e., the replica g with the lowest cost $\delta_{r,g}$ in the topology). Both GET-NEXT and GET-PREV (line 26) involve the closest serializer for key k's leaf node and the serializer of next or previous key's leaf node. An UPDATE (line 30) must involve all the serializers of the leaf node that contains key k.

An INSERT (line 32) can alter the tree structure causing one or more nodes to split. Such splits can propagate to parent nodes up to the root in the worst case. For consistency, a modification in the tree structure should involve the serializers of every tree node modified as part of the insert. A conservative solution would be to include the serializers of the leaf node and of its ascendants in the tree up to the root. We introduce an optimization to reduce the number of serializers involved in the insert. We initially check if the leaf node would split as a result of the insert, i.e., if the leaf size is close to the maximum allowed (line 34). If so, we include the serializers of the leaf parent in the list of serializers of the insert. We then repeat the procedure above for each of the ascendents of the leaf up to the root.

A replica invokes primitive serialize(op) to serialize request op. The serialization of op involves all the serializers in $op.dst$, using the procedure described in §II-B. Once the request is serialized, it is available to a replica by means of primitive deliver(op).

When a replica delivers a request op (line 10), it checks whether the current serializers of op match the serializers used to order op. This is necessary because GeoPaxos+ periodically recomputes a "new configuration" based on the workload (described in §IV-C). If the configuration has changed between the serialization of a request and its delivery, a replica may need to resubmit the request. In particular, the replica resubmits the request if the new set of serializers for op include a serializer that was not used to serialize op. A new configuration request is ordered with respect to all requests it shares objects with; thus, replicas observe a consistent mapping of objects to serializers.

C. Distributed B+Tree management

GeoPaxos+ keeps track of the objects read and written as part of the execution of a request. These are counter fields *reads* and *writes* of a tree node (in Algorithm 1, these counters are updated at lines 43 and 45, respectively). Note that the read and write counters are per replica, in particular, the replica that received client's request op (stored in $op.from$).

As clients issue requests, the workload information for tree nodes builds up. To recompute a configuration, using the mechanism presented in §III, a replica invokes the DOP function, as depicted in Algorithm 1, line 58. Since the operation impacts many tree nodes, the destination comprises

Algorithm 1: Replica r implements distributed B$^+$Tree $\mathcal{T}$.

```
Initialization:
   $\mathcal{T} \leftarrow \varnothing$                                        {local B+Tree replica}
when receive(OP, $k, v$) from a client
   $op.type \leftarrow$ OP
   $op.k \leftarrow k$
   $op.v \leftarrow v$
   $op.from \leftarrow r$
   $op.dst \leftarrow$ psite($op$)
   serialize($op$)
when deliver($op$)                                  {delivers ordered operation op}
   if psite($op$) $\subseteq op.dst$ then
      $result \leftarrow$ execute($op$)
      send $result$ to client
   else
      $op.dst \leftarrow$ psite($op$)
      serialize($op$)
function psite($op$)                                {returns the serializers of request op}
   if $op.type =$ DOP then
      $dst \leftarrow \{g_1, ..., g_N\}$
   else
      $dst \leftarrow \emptyset$
      $n \leftarrow$find($\mathcal{T}, op.k$)
      switch $op.type$
         case READ :
            $dst \leftarrow g \mid g \in n.reps \wedge \forall h \in n.reps, \delta_{r,g} \leq \delta_{r,h}$
         case GET-NEXT $\vee$ GET-PREV :
            $dst \leftarrow g \mid g \in n.reps \wedge \forall h \in n.reps, \delta_{r,g} \leq \delta_{r,h}$
            $n \leftarrow op.type(\mathcal{T}, op.k, \perp)$
            $dst \leftarrow dst \cup g \mid g \in n.reps \wedge \forall h \in n.reps, \delta_{r,g} \leq \delta_{r,h}$
         case UPDATE :
            $dst \leftarrow n.reps$
         case INSERT :
            $dst \leftarrow dst \cup n.reps$
            while $n.parent \neq \varnothing \wedge n.size + 1 = \mathcal{T}.order$ do
               $n \leftarrow n.parent$
               $dst \leftarrow dst \cup n.reps$
   return $dst$
function execute($op$)                              {executes the delivered operation}
   if $op.type \neq$ DOP then
      $n \leftarrow$find($\mathcal{T}, op.k$)
      while $n.parent$ do                           {update R/W counter for replica op.from}
         if $op.type \in \{$READ, GET-NEXT, GET-PREV$\}$ then
            $n.reads[op.from] \leftarrow n.reads[op.from] + 1$
         else
            $n.writes[op.from] \leftarrow n.writes[op.from] + 1$
         $n \leftarrow n.parent$
      execute $op.type(op.k, op.v)$ using tree $\mathcal{T}$
   else
      foreach $n \in \mathcal{T} \mid n.changed =$ true                   {if node n changed}
         $replicas \leftarrow \varnothing$
         $cost = \infty$
         foreach $i \in \mathcal{P}_{n,i}$                          {for each configuration}
            if $cost < \mathcal{C}_{n,i}$ then                     {updates mininum cost}
               $cost \leftarrow \mathcal{C}_{n,i}$
               $replicas \leftarrow i$
         $n.reps \leftarrow replicas$                         {minimum cost configuration}
         $n.changed \leftarrow$ false                           {resets changed flag}
function dop()                                      {recomputes configuration}
   $op.type \leftarrow$ DOP
   $op.dst \leftarrow$ psite($op$)
   serialize($op$)
```

all replicas. Once delivered, replicas apply the calculations explained in §III-B to each node n that has changed since the last time the computation happened. After calculating the cost for each configuration $\mathcal{P}_{n,i}$ according to (1), replica r updates node n's serializers to $\mathcal{P}_{n,best}$ as defined in (2).

V. Implementation

We extended GeoPaxos with the techniques described in the paper. The C++ source code is publicly available.[1] In GeoPaxos+, each replica is a multi-threaded process with a full copy of the B+Tree, with the additional fields described in Table II and initially empty. A set of threads receives requests from clients and enqueues such requests for further processing. Additional threads the serializers for each request *op* in this queue invoking the function `psite(`*op*`)` before propagating the operation to the corresponding groups. A learner for each Paxos instance is executed as an independent thread and only synchronizes with other learners when a request multiple serializers. Multi-threaded clients connect to the closest replica and submit requests in a closed-loop, i.e., a new request is only submitted after the reply for the current one.

VI. Performance evaluation

In this section, we explain our setup (§VI-A), experimentally assess GeoPaxos+ in a LAN and WAN (§VI-B and §VI-C), evaluate the performance of our optimization model (§VI-D), and conclude with a summary of our findings (§VI-E).

A. Evaluation rationale

We implemented requests to retrieve a key-value entry from the tree and to insert 100-byte objects in the tree; hereafter, we refer to them as *read* and *write* operations, respectively. We evaluate GeoPaxos+ under three client access patterns: (i) *locality workload* represents a situation where co-located clients tend to access more often the same subset of the system objects, which is common in typical geographically distributed applications (e.g., social networks); (ii) *uniform workload* represents a situation without locality: each client has the same probability of accessing any key in the system; (iii) *skewed workload* corresponds to a situation with moderate locality, where some objects have higher probability of being accessed by every client in the system, corresponding to a Zipfian distribution. We also vary the percentage of reads in the workload.

We conducted experiments in a LAN and a WAN. The LAN provides a controlled environment, where experiments can run in isolation; the WAN represents a setting in which we expect our solution to be used in practice. In a LAN, we use a cluster of nodes, each node with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8GB of memory, sata SSDs and 1Gbps ethernet cards. The WAN deployment runs on Amazon EC2, where each node is a m3.large instance with 2vCPUs and 7.5GB of memory. We deploy replicas along 3 Amazon geographic regions (i.e., Central Europe, California and Virginia). Paxos acceptors for the same replica (i.e., used to replicate serialization logic, see §II) reside in different regions to tolerate the catastrophic event of a complete geographic region failure. We report results for our B+Tree protocol with a random assignment of serializers ("DBT_b" in the graphs) and after the computation of our optimization model ("DBT_a").

We compare our replicated B+Tree to implementations of a B+Tree using Multi-Paxos [18] and EPaxos [3]. Multi-Paxos stands as a baseline for performance reference while EPaxos poses as a more realistic competitor since, similarly to our protocol, it allows clients to connect to the closest replica, being more judicious about round-trips in geographically distributed environments.

B. Performance in a LAN

The first experiment evaluates GeoPaxos+ under three different access patterns and two read percentages. We consider settings with 3 and 5 replicas. In Figure 5(a) we can observe that with locality (top left graph) the read percentage has low impact on performance; GeoPaxos+ outperforms Multi-Paxos and EPaxos in each workload, and scales with the number of replicas, reaching over 140 thousand requests per second in the best case against around 60 thousand requests per second for EPaxos. These results reinforce our initial expectations that with locality, few objects tend to be assigned to more than one serializer, resulting in both read and write requests involving a single serializer. With skewed and uniform workloads, the read percentage matters since the greater the number of write operations, the higher the chances of multi-serializer requests. GeoPaxos+ outperforms the competitors with more than 90% of reads executing up to 150 thousand requests each second.

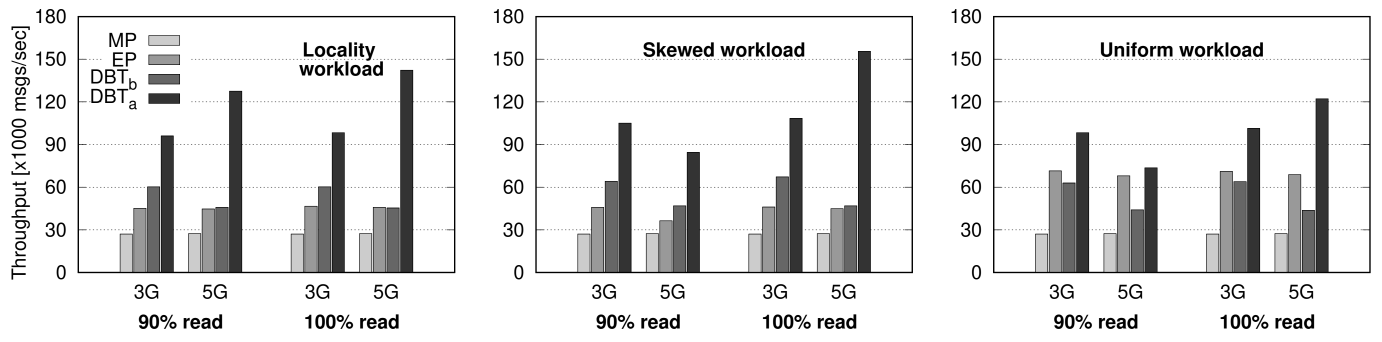
Figure 5(b) assesses GeoPaxos+ with up to 9 replicas. With read-only operations, GeoPaxos+ scales with the number of replicas despite the different workloads. In particular, for the uniform workload, the worse results for 7 and 9 replicas are due to reaching our cluster maximum capacity before we could reach peak performance (i.e., we ran out of resources for clients). As we add write operations, the dependency on the workload becomes more evident. With locality, the system throughput behaves similarly independent of the operation type. With skewed and uniform workload, the additional replicas do not bring performance improvements.

C. Performance in a WAN

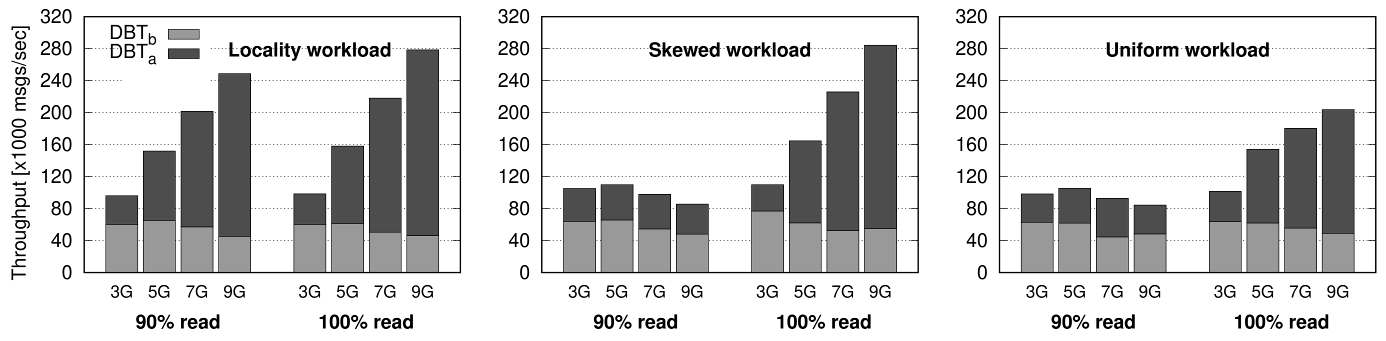
The following experiments assess GeoPaxos+ and EPaxos in a disaster-tolerant deployment within three Amazon regions, two in the US and one in Europe. The latencies vary from approximately 70 msec for the two regions in the US, and 90 and 150 msec from US regions to Europe.

Figure 6 exhibits the latency observed by clients in Europe for each workload, with 95% and 100% read operations in 3 different situations: (i) *low contention* represents a scenario with 54 clients equally distributed among all the regions; (ii) *medium contention* presents 5400 clients; and (iii) *high contention*, 16200 clients. With low contention in locality and

[1] https://github.com/paulo-coelho/gbtree



(a) Varying workload and read percentage with 3 and 5 replicas.



(b) Effect of dynamic access on throughput with increasing number of replicas.

Fig. 5: Peak throughput in a LAN ("DBT_b": random assignment of serializers, "DBT_a": after optimization model).

uniform workloads, the latency for each protocol is similar to the expected optimum: one round-trip to the closest region (around 90 msec). With a skewed workload, EPaxos has a greater latency since the chance of conflicts increase due to most clients accessing the same subset of keys.

With medium and high contention, EPaxos latencies soar, reaching above 2 seconds in some cases, since more clients means more conflicts and additional round-trips (slow path). Besides, we observed that EPaxos has higher CPU usage as the conflict graph grows, also contributing to such higher latencies. Although GeoPaxos+'s latency also increases with the number of clients, it is not as noticeable as EPaxos's.

Figure 7 shows that GeoPaxos+'s throughput ranges from around 40000 operations per second in the worst case (95% reads and skewed workload) to over 85000 with read-only operations and locality workload. EPaxos presents throughput below 3000 operations per second for skewed and uniform workloads, and a best-case of around 20000 read-only operations per second with the locality workload.

D. *Performance of optimizer*

The last set of experiments evaluate the performance of the optimizer. The experiments measure the average time to execute the optimization algorithm for a B$^+$Tree with one million keys considering four different strategies: (i) *brute force*: the algorithm is executed for each tree node; (ii) *hot replicas*: the algorithm is only executed when the number of accesses from clients in a particular replica reaches a threshold of the number of accesses for the most accessed replica (10% in our case); (iii) *caching*: we save results from previous calculations to be used by similar workloads; and finally, (iv) *hot replicas + caching*: we combine strategies (ii) and (iii). The tree is initially half populated, then we compute the serializers, and reset the read-write counters. Then, clients execute with a locality workload and write ratio of 50%.

Figure 8(a) shows the time for each strategy for 3, 4 and 8 replicas. The fastest technique is represented by the combination of hot replicas and caching strategies. In particular, such a combination can reduce optimization time by up to 50 times when compared to the brute force strategy. The last experiment, depicted in Figure 8(b), exhibits the combined technique with respect to using only the caching (LRU) strategy. Even after a few iterations, the time for "Hot replicas + LRU" is around half the time of LRU alone.

E. *Summary*

The results from the LAN deployment allow us to draw the following conclusions:

- GeoPaxos+ outperforms both MultiPaxos and EPaxos. GeoPaxos+'s advantage stems from its efficient ordering mechanism and the multithreaded execution of requests that the ordering mechanism enables.
- Differently from MultiPaxos and EPaxos, GeoPaxos+ scales performance (throughput) with an increasing number of replicas.

Regarding the WAN deployment, our results show that:

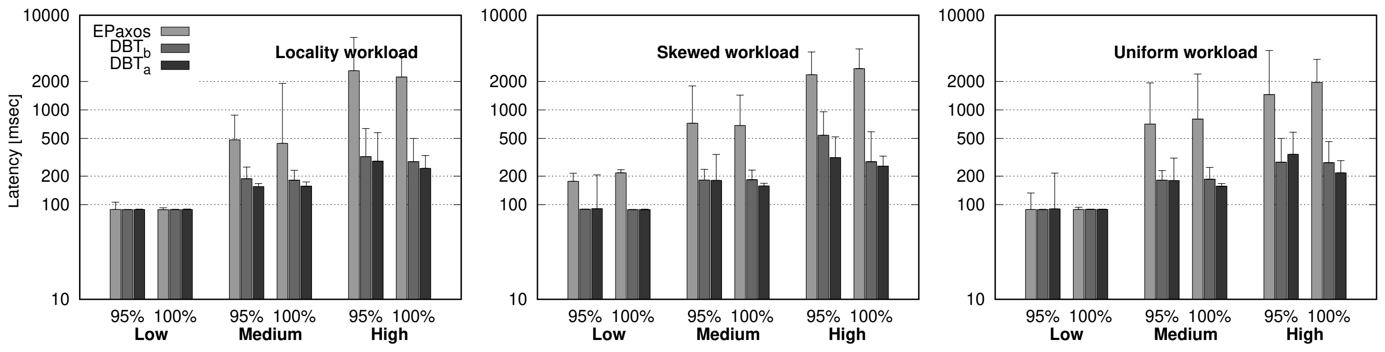


Fig. 6: Mean latency in a WAN (whiskers represent 95^{th} percentile).

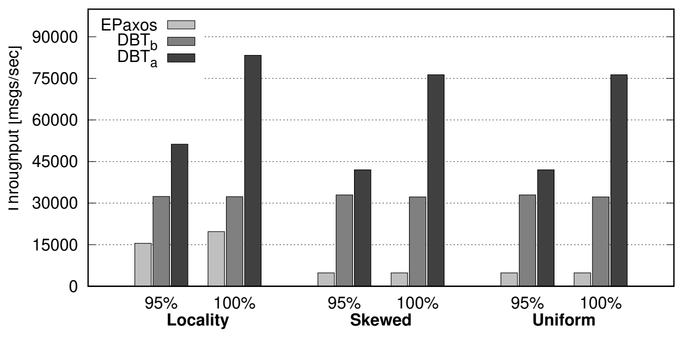
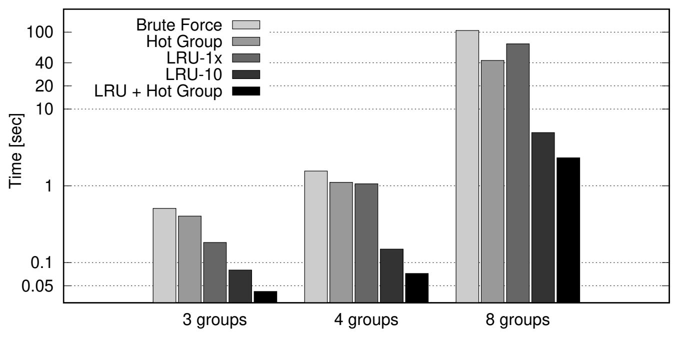
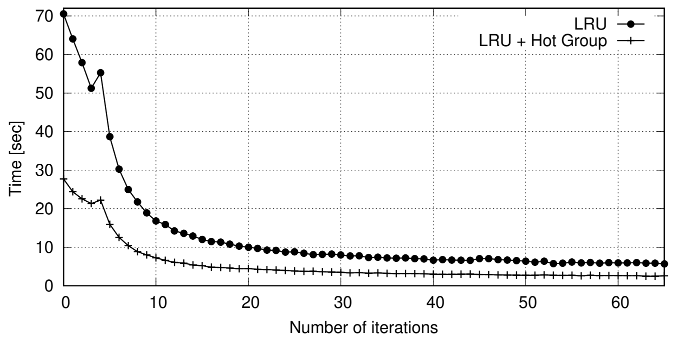


Fig. 7: Peak throughput in a WAN.



(a) Time to define object access. LRU-1x and LRU-100x represent executions with the caching technique after 1 and 100 runs.



(b) Time for LRU vs. "LRU + Hot replicas".

Fig. 8: Dynamic object access duration for different techniques with 1 million objects.

- GeoPaxos+ outperforms EPaxos in terms of latency for low, medium, and high contention with every workload.
- Taking clients' access patterns into consideration improves GeoPaxos+ throughput more than twofold in read-only scenarios, no matter the workload.
- GeoPaxos+'s throughput is 3 to 20 times greater than EPaxos's, because of our dynamic object placement and the processing cost of EPaxos conflict graph as contention builds up.

Concerning the optimizations in the dynamic object replacement algorithm, the results show their effectiveness with a reduction in execution time of almost two orders of magnitude.

VII. Related work

If on the one hand state machine replication is widely used to increase service availability (e.g., [19], [20], [1]), on the other hand it is often criticized for its overhead. From single-leader algorithms (e.g., Paxos [21]) to leaderless algorithms (e.g., [3], [22]) and variations that take the semantics of operations into account (e.g., [23], [24]), all efforts have been directed at finding faster ways to order operations. None of the solutions, however, can avoid the latency imposed by a geographically distributed majority quorum of replicas [25]. Furthermore, existing solutions experience reduced performance as the number of replicas increases.

GeoPaxos+ improves the performance of state machine replication by (a) exploiting the fact that operations do not need a total order; (b) distinguishing ordering from execution; (c) judiciously choosing the serializer (i.e., Paxos group) that will order operations; (d) dynamically analyzing and recomputing serializers to account for locality; and (e) differentiating read and write operations to increase parallelism. Partially ordering operations with the goal of improving performance has been previously implemented by EPaxos, Alvin, Caesar and M2Paxos.

EPaxos [3] improved on traditional Paxos [21] by reducing the overload on the coordinator and allowing any replica to order operations. As long as replicas observe common dependencies set, operations can be ordered in one round-trip fast decision).

Alvin [22] is a system for managing concurrent transactions that relies on Partial Order Broadcast (POB) to order conflicting transactions. While POB is similar to EPaxos, its main contribution lies in the substitution of EPaxos complex dependency graph analysis by a set of cycle-free dependencies based on timestamps.

Caesar [26] extends POB and reduces the number of scenarios that would impose one additional round-trip (slow decisions). Differently from Alvin and EPaxos, where nodes must agree on operation dependency sets, Caesar seeks agreement on a common final timestamp for each operation. This strategy allows fast decisions even in specific cases where dependencies do not match, resulting in better performance as contention increases. Depending on the workload, however, the rate of slow decisions in EPaxos, Alvin and Caesar increases, a price to be paid by geographically distributed applications.

M2Paxos [4] is another implementation of Generalized Consensus [23] that does not establish operation dependencies based on conflicts, but, similar to GeoPaxos+, maps replicas to accessed objects. M2Paxos guarantees that operations that access the same objects are ordered by the same replica. It needs at least two communications steps for local operations and one additional step for remote operations. M2Paxos's mechanism to handle operations that access objects mapped to multiple replicas requires remapping the involved objects to a single replica. Replicas executing different operations may dispute the same objects indefinitely.

Mencius [27] extends traditional Multi-Paxos with a multi-leader solution that partitions the sequence of consensus instances among geographically distributed replicas to avoid the additional round-trip for clients far from the single-leader. Besides, it provides a mechanism to deal with unbalanced load, allowing one replica to propose a `SKIP` message when there is no operation to be executed in that instance. Mencius also allows out-of-order execution of commutative operations.

Several solutions that partition (i.e., shard) the data have appeared in the literature. Systems in this category are sometimes referred to as partially replicated systems, as opposed to designs in which each replica has a full copy of the service state, like in GeoPaxos+. Spanner [1] is a partitioned distributed database for WANs. It uses a combination of two-phase commit and a TrueTime API to achieve consistent multi-partition transactions. TrueTime uses hardware clocks to derive bounds on clock uncertainty, and is used to assign globally valid timestamps and for consistent reads across partitions. It requires elaborate synchronization mechanisms to keep the clock skew among nodes within an acceptable limit. Furthermore, Spanner supports a more restrictive type of operations (read-write objects) than state machine replication (read-modify-write objects).

Spinnaker [28] is similar to the approach presented here. It also uses several instances of Multi-Paxos to achieve scalability. However, Spinnaker does not support operations across multiple Multi-Paxos instances.

Differently from existing sharded systems, where replicas contain only part of the service state, in GeoPaxos+ each replica contains the entire state. In doing this, we can improve performance without sacrificing the simplicity of the state machine replication approach. Moreover, there is no need to reshard and migrate data across server nodes for load balance or in response to failures.

Regarding distributed B$^+$Trees, HyperDex [29] and Yesquel [30] are the most related to our proposal. HyperDex implements a partitioned key-value store which allows efficient search functions and secondary indexes based on a novel multi-dimensional hash function. Yesquel implements a distributed B-tree and proposes several optimizations to use the tree for a distributed SQL database. The architecture and concurrency control used in Yesquel are similar to Sinfonia [31] mini-transactions. Two other designs of a distributed B$^+$Tree were proposed in Minuet [32], on top of Sinfonia [31], and STI-BT [33]. Both exploit multi-versioning to enhance concurrency between transactions. While Minuet handles multi-versioning externally to Sinfonia, relying on a centralized snapshot identifier that increments whenever a read-only transaction requires a fresh snapshot, STI-BT uses a scalable distributed multi-versioning scheme to improve the performance of read-only transactions. Besides, Minuet distributes the tree nodes randomly across the Sinfonia cluster. STI-BT exploits the structure of the tree to co-locate tree nodes and maximize data locality. Since these solutions partition the B$^+$Tree state, they also require resharding with data migration to adapt to locality changes. Besides, Yesquel largely relies on data stored within the client for performance. Our B$^+$Tree built on GeoPaxos+ is transparent to clients.

The work in [34] describes a workload-driven approach that optimizes the latency of a sharded database by leveraging leader placement, the roles of different servers, and replica location. GeoPaxos+ is more general, both with respect to the metrics to be optimized and the particular application.

Droopy and Dripple [35] introduce an approach similar to GeoPaxos+ with a dynamic set of serializers and a mechanism to cope with workload changes. Like Droopy, our solution adapts to workload changes and can redefine the set of serializers to reduce latency. Unlike Droopy, we monitor the access to objects to dynamically change the serializers. Unlike Dripple, we do not partition the state, but the ordering. Besides, we do not require either grouping non-commutative requests in advance or additional computation to break cycles and define the final execution order.

VIII. Final remarks

Online services with clients distributed all over the globe are becoming common, imposing tough requirements on replicated protocols. Coordinating replicas geographically distributed, while keeping performance at acceptable levels is challenging. The paper proposes a practical geographical state machine replication protocol and illustrates its use with a B$^+$Tree. Although the contribution focuses on a B$^+$Tree, the discussion is general enough to be used with other complex data structures. Most important, we describe the challenges that one must overcome to design general services in the context of state machine replication.

References

[1] J. D. J. C. Corbett and M. E. et al, “Spanner: Google’s globally distributed database,” in *OSDI*, 2012.

[2] N. Schiper, P. Sutra, and F. Pedone, “P-Store: Genuine partial replication in wide area networks,” in *SRDS*, 2010.

[3] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is more consensus in egalitarian parliaments,” in *SOSP*, 2013.

[4] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, “Making fast consensus generally faster,” in *DSN*, 2016.

[5] T. Kraska, G. Pang, M. J. Franklin, and S. Madden, “MDCC: Multi-Data Center Consistency,” *CoRR*, 2012.

[6] P. Coelho and F. Pedone, “Geographic state machine replication,” in *SRDS*, 2018.

[7] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, “Transactional storage for geo-replicated systems,” in *SOSP*, 2011.

[8] A. Brodersen, S. Scellato, and M. Wattenhofer, “Youtube around the world: Geographic popularity of videos,” in *WWW*, 2012.

[9] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *CACM*, vol. 21, pp. 558–565, July 1978.

[10] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[11] T. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[12] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[13] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *SIGMOD*, 1984.

[14] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty processor,” *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[15] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *Trans. on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990.

[16] P. Coelho and F. Pedone, “Geographic state machine replication,” tech. rep., USI, 2017.

[17] D. K. Gifford, “Weighted voting for replicated data,” in *SOSP*, 1979.

[18] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: an engineering perspective,” in *PODC*, 2007.

[19] B. e. a. Calder, “Windows azure storage: A highly available cloud storage service with strong consistency,” in *SOSP*, 2011.

[20] P. Hunt, M. Konar, F. Junqueira, and B. Reed, “ZooKeeper: Wait-free coordination for Internet-scale systems,” in *USENIX ATC*, 2010.

[21] L. Lamport, “The part-time parliament,” *Trans. on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[22] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran, “Be general and don’t give up consistency in geo-replicated transactional systems,” in *OPODIS*, 2014.

[23] L. Lamport, “Generalized consensus and paxos,” Tech. Rep. MSR-TR-2005-33, Microsoft Research (MSR), Mar. 2005.

[24] F. Pedone and A. Schiper, “Generic broadcast,” in *DISC*, 1999.

[25] L. Lamport, “Lower bounds for asynchronous consensus,” *Distributed Computing*, vol. 19, no. 2, pp. 104–125, 2006.

[26] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran, “Speeding up consensus by chasing fast decisions,” in *DSN*, 2017.

[27] Y. Mao, F. Junqueira, and K. Marzullo, “Mencius: Building efficient replicated state machine for WANs,” in *OSDI*, 2008.

[28] J. Rao, E. Shekita, and S. Tata, “Using Paxos to build a scalable, consistent, and highly available datastore,” in *VLDB*, 2011.

[29] R. Escriva, B. Wong, and E. G. Sirer, “Hyperdex: A distributed, searchable key-value store,” in *SIGCOMM*, 2012.

[30] M. K. Aguilera, J. B. Leners, and M. Walfish, “Yesquel: scalable sql storage for web applications,” in *SOSP*, 2015.

[31] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, “Sinfonia: a new paradigm for building scalable distributed systems,” in *SOSP*, 2007.

[32] B. Sowell, W. Golab, and M. A. Shah, “Minuet: A scalable distributed multiversion b-tree,” in *VLDB*, 2012.

[33] N. Diegues and P. Romano, “Sti-bt: A scalable transactional index,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2408–2421, 2015.

[34] A. Sharov, A. Shraer, A. Merchant, and M. Stokely, “Take me to your leader! online optimization of distributed storage configurations,” in *VLDB*, 2015.

[35] S. Liu and M. Vukolić, “Leader set selection for low-latency geo-replicated state machine,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1933–1946, 2016.