Keeping up with storage: Decentralized, write-enabled dynamic geo-replication

Pierre Matri, María S. Pérez, Alexandru Costan, Luc Bougé, Gabriel Antoniu

Keeping up with Storage: Decentralized, Write-Enabled Dynamic Geo-Replication

Pierre Matri[a], María S. Pérez[a], Alexandru Costan[c], Luc Bougé[1], Gabriel Antoniu[d]

[a] *Ontology Engineering Group, Universidad Politécnica de Madrid, Spain*
[b] *IRISA / INSA Rennes, France*
[c] *IRISA / ENS Rennes, France*
[d] *INRIA Rennes, France*

Abstract

Large-scale applications are ever-increasingly geo-distributed. Maintaining the highest possible *data locality* is crucial to ensure high performance of such applications. Dynamic replication addresses this problem by dynamically creating replicas of frequently accessed data close to the clients. This data is often stored in decentralized storage systems such as Dynamo or Voldemort, which offer support for *mutable data*. However, existing approaches to dynamic replication for such mutable data remain centralized, thus incompatible with these systems. In this paper we introduce a write-enabled dynamic replication scheme that leverages the decentralized architecture of such storage systems. We propose an algorithm enabling clients to locate tentatively the closest data replica without prior request to any metadata node. Large-scale experiments on various workloads show a read latency decrease of up to 42% compared to other state-of-the-art, caching-based solutions.

Keywords: cloud, replication, geo-replication, storage, fault-tolerance, consistency, database, key-value store

1. Introduction

Large-scale applications such as social networks are being increasingly deployed over multiple, geographically distributed datacenters (or *sites*). Such geo-distribution provides fast data access for end-users worldwide while improving fault-tolerance, disaster-recovery and minimizing bandwidth costs. Today's cloud computing services [1, 2] allow a wider range of applications to benefit from these advantages as well. However, designing geo-distributed applications is difficult due to the high and often unpredictable latency between sites [3].

Such geo-distributed applications span a large range of specific use-cases. For instance, cientific applications such as the MonALISA [4] monitoring system for the CERN LHC Alice experiment [5]. This application collects and aggregates monitoring data from 300+ sites distributed across the world, that must be delivered to scientists worldwide in real-time. The users of commercial applications, such as Facebook, ever-increasing amounts of data that needs to be accessible worldwide. Ensuring the lowest possible access time for users is crucial for the user experience.

A key factor impacting the performance of such applications is *data locality*, *i.e.* the location of the data relatively to the application. Accessing remote data is orders of magnitude slower than using local data. Although such remote accesses may be acceptable for rarely-accessed data (*cold data*), they hinder application performance for frequently-used data (*hot data*). For instance, in a social network application, popular profiles should be replicated at all sites whereas others can remain located at fewer locations. Finding the right balance between replication and storage is critical: replicating too many profiles wastes costly memory and bandwidth, while failing to replicate popular ones results in degraded application performance.

Dynamic replication [6] proposes to solve this issue by dynamically replicating hot data as close as possible to the applications that access it. This technique is leveraged in Content Delivery Networks (*CDN*) to cache *immutable* data close to the final user [7, 8]. Similarily, it

Email addresses: pmatri@fi.upm.es (Pierre Matri), mperez@fi.upm.es (María S. Pérez), alexandru.costan@irisa.fr (Alexandru Costan), luc.bouge@ens.ens-rennes.fr (Luc Bougé), gabriel.antoniu@inria.fr (Gabriel Antoniu)

is used in storage systems such as GFS [9] or HDFS [10] to replicate *mutable* data, by relying on the *centralized* metadata management of these systems [11, 12]. Yet, such an approach contradicts the design principles of *decentralized* storage systems such as Dynamo [13] or Voldemort [14], which aim to enable clients to locate data without exchanges with dedicated metadata nodes.

Furthermore, handling *mutable* objects in the context is difficult. Indeed, the dynamic replicas have to be kept synchronized with the origin data, without impacting the consistency guarantees of the underlying system. To the best of our knowledge, no decentralized, write-enabled dynamic replica location and management method exists in the literature today. Reaching this goal while providing predictable overhead and guaranteed accuracy is not trivial. In this paper we demonstrate that this objective can be reached by combining the architecture of this systems with deceptively simple algorithms from the literature. We make these contributions, which substantially revise and extend the early principles we previously introduced in [15]:

- After briefly introducing the applications we target (Section 2), the related work (Section 3) and the storage systems we target (Section 4), we **characterize the challenges of decentralizing write-enabled dynamic data replication** (Section 5).
- We address these challenges with **a decentralized data popularity measurement scheme** (Section 6), which leverages existing state-of-the-art storage system architecture to identify hot data cluster-wide dynamically.
- Based on these popularity measurements, we describe **a dynamic data replication algorithm** which dynamically creates and manages replicas of hot data as close as possible to the applications (Section 7).
- We enable clients to locate the closest of such data replicas using **an approximate object location method** (Section 8), which minimizes storage latency by avoiding communication with any dedicated metadata node.
- We develop **a prototype implementation leveraging the above contributions**, integrated with the Voldemort distributed key-value store (Section 9), and prove the effectiveness of our approach with **a large-scale experimental study** on the Amazon Cloud (Section 10). We observe a read latency decrease of up to 42% compared to other state-of-the-art, caching-based algorithms.

We discuss the effectiveness and applicability of our contribution (Section 11), and conclude on future work that further enhances our proposal (Section 12).

2. Large-scale, data-intensive applications

In this paper we target large-scale applications serving large amounts of data to users around the world, while seeking to enable low-latency access for these users to potentially-mutable data. Examples of such applications span multiple use-cases, among which:

Scientific system monitoring. Monitoring a geo-distributed cluster requires collecting potentially large number of metrics from computers around the world. This is for example the case for MonALISA [4], monitoring thousands of servers distributed in more than 300 sites around the world. The collected data is aggregated live, and is used to provide live insights about the system performance and availability around the world. To deliver the real-time promise of MonALISA, ensuring that the monitoring data that is needed by scientists around the world is located as close as possible to them is crucial.

Social networks. Business applications such as social networks ingest overwhelming amounts of data. Facebook, for example, is expected to react 2 billion active profiles in the next few weeks [16]. Every single day, it processes 350 million photo uploads [17] or 6 billion posts [18]. The strong 500:1 read to write ratio [19] calls for large-scale caching. However, some of this data is mutable by nature. This is for example the case of user profiles, which are hard to cache while keeping this cache synchronized with the source data, calling for heavyweight, custom cache invalidation pipelines [20].

The real-time promise of such applications requires to keep up-to-date data as close as possible to the end-user. The global scale of these applications makes this difficult while keeping costly bandwidth and storage usage low. We will detail these challenges in Section 5. These challenges are independent of the type of platforms such as compute grids for MonALISA or clouds for Facebook.

3. Related work

In the literature, dynamic replication stands as a topic of interest for all applications requiring access to shared

data from many geo-distributed locations. Most of these contributions can be classified in two categories:

Immutable data, decentralized management. A range of applications require to provide their users with fast and timely access to static resources such as images or videos. This is the case of most global internet applications, in which Content Distribution Networks (*CDNs*) help provide a good user experience by creating replicas of *static, immutable* data as close as possible to the clients that access it.

Yet, CDNs are targeted at *serving content directly to the final user*. In this paper, we focus on *allowing a geo-distributed application to access a geo-distributed data source* with the lowest possible latency.

Kingsy Grace *et al.* [7] provide an extensive survey of replica placement and selection algorithms available in the literature. Among these, Chen *et al.* [8] propose a dissemination-tree based replication algorithm leveraging a peer-to-peer location service. Dong *et al.* [21] transform the multiple-location problem into several classical mathematical problems with different parameter settings, for which efficient approximation algorithms exist. However, they don't consider the impact of replication granularity on performance and scalability. Wei *et al.* [22] address this issue by developing a model to express availability as a function of replica number. This approach, however, only works within a single site, as it assumes uniform bandwidth and latency, which is not the case with the geo-distributed workloads that we target. Inspired by the P2P systems, [23] proposes an adaptive decentralized file replication algorithm that achieves high query efficiency and high replica utilization at a significantly low cost. In [24], MacCormick *et al.* enable storage systems to achieve balanced utilization of storage and network resources in the presence of failures, and skewed distributions of data size and popularity. Madi *et al.* [25] consider a wider usage of parameters in the context of data grids such as read cost or file transfer time.

Mutable data, centralized management. However, a range of applications rely on mutable data. This is for example the case in MonALISA monitoring of the CERN LHC experiment [4, 5]. In a web applications, this is observed with social network profile pages, status pages, comments on a news thread, or more generally services displaying publicly user-generated content. In all these applications, we also observe that the data objects are mutable (changing aggregates from new monitoring events, user updating their profile, posting new statues or comments). Available geo-replications solutions available typically either require the application to explicitly clear modified objects from distant caches, or leverage a centralized replication manager that contradicts the decentralized design of most state-of-the-art, geo-distributed data stores. Dynamic replication enables the geo-distributed storage system to replicate in near real-time the most requested objects as close as possible to the application instances accessing them.

Efficiently creating and placing replicas of hot data is not enough. Indeed, one needs to ensure as well that those replicas are kept in synchronization with the original data. This is usually the case of applications relying on a globally-distributed file system. Overall, the proposed solutions in the literature leverage the centralized metadata management of certain storage systems such as HDFS [10] or GFS [9] to allow the clients to locate the closest available replica of the data they want to access. In that context, Ananthanarayanan *et al.* [11] propose a popularity-based dynamic replication strategy for HDFS aimed at improving the performance of geo-distributed Map-Reduce clusters. Jayalakshmi *et al.* [12] models a system designed to direct clients to the most optimal replica available.

Our proposal enables writes to any given object in a decentralized, large-scale storage system to be transparently forwarded to the existing dynamic replicas of that object, without requiring explicit cache eviction requests from the application. In contrast, replication strategies adopted in CDNs such as Dynamic Page Caching [26] have a substantially different target ; they focus on offering fine-grained caching based on configured user request characteristics (cookies, request origin, ...), while still accessing the origin data replica for dynamic, mutable objects.

Replica selection algorithm Targeted work on replica selection prove that adopting a relevant data location algorithm can lead to significant performance improvements. Mansouri *et al.* [27] propose a distributed replication algorithm named Dynamic Hierarchical Replication Algorithm (DHR), which selects replica location based on multiple criteria

such as data transfer time and request–waiting-time. Kumar *et al.* [28] address the problem of minimizing average query span, *i.e.* the average number of machines that are involved in the processing of a query through co-location of related data items. C3 [29] goes even further by dynamically adapting replica selection based on real-time metrics in an adaptive replica selection mechanism that reduces request tail latency in presence of service-time fluctuations in the storage system.

However, none of these contributions considers the case of *mutable data* stored in *decentralized data stores*, such as Cassandra [30] or Voldemort [14]. Facebook, for example, circumvents the issue by directing all write requests to a single data center and using a dedicated protocol to keep the cache consistent across other regions [19]. In this paper, we fill this gap by enabling efficient data replication of mutable data in *geo-distributed, decentralized* data stores.

4. Background: The systems we target

Let us first briefly describe the key architectural principles that drive the design of a number of decentralized systems. Dynamo [13] has inspired the design of many of such systems, such as Voldemort [14], Cassandra [30] or Riak [31]. In this paper we target this family of systems, which are widely used in the industry today.

Data model. Dynamo is a *key-value store*, otherwise called *distributed associative array*. A key-value store keeps a collection of *values*, or *data objects*. Each object is stored and retrieved using a *key* that uniquely identifies it.

DHT-based data distribution. Objects are distributed across the cluster using consistent hashing [32] based on a *distributed hash table* (*DHT*), as in Chord [33]. Given a hash function $\text{H}(x)$, the output range $[h_{min}, h_{max}]$ of the function is treated as a circular space (h_{min} sticking around to h_{max}), or *ring*. Each node is assigned a different random object within this range, which represents its position on the ring. For any given key k, a position on the ring is determined by the result of $\text{H}(k)$. The *primary node* holding the primary *static replica* of the object is the first one encountered while walking the ring passed this position. To ensure fault-tolerance, additional static replicas are created at the time the object is stored. These are placed on the next r nodes following the primary node on the ring, r being the configured replication factor of the system (usually 2).

P2P cluster state dissemination. The position of each node on the ring is advertised in the cluster using a family of *peer-to-peer* (*P2P*) protocols: *Gossip* [34]. Each node periodically disseminates its status information to a number of randomly-selected nodes and relays status information received from other nodes. This method is also used to detect and advertise node failures across the cluster [35].

Client request routing. By placing objects deterministically in the cluster, Dynamo obviates the need for dedicated metadata servers. Clients are able to perform *single-hop* reads, *i.e.* address their requests directly to the nodes holding the data. This enables a minimal storage operation latency and higher throughput. Should a client address the request to a node not holding the requested data, this node will forward the request directly to the correct one. This correct node is determined using the ring state information disseminated throughout the cluster.

Deterministically placing data objects and disseminating ring status in the cluster enables each node to route incoming client requests directly to a node holding the data. Operation latency is further reduced by opening cluster state information to the clients so they can address their requests straight to the correct node, without any metadata server involved.

We found our strategy on these design principles, which allow us to guarantee the correctness of the proposal we describe in this paper. We choose not to modify the original static replication mechanism, offering the same data durability as the underlying system. We also do not change the server-side client routing mechanism, consequently guaranteeing that static replicas are always reachable. This allows us to focus on developing an efficient *heuristic* that maximizes the accuracy of popular object identification, optimizes the creation and placement of dynamic replicas of such popular objects, and helps clients efficiently locating the closest of these replicas.

5. Our proposal in brief: outline and challenges

In this paper, we demonstrate that it is possible to integrate dynamic replication with the existing architec-

ture of these storage systems, which enables us to leverage their existing, built-in algorithms to efficiently handle read and writes in geo-distributed environments.

Such dynamic replication seeks to place new copies of the hot data in sites as close as possible to the application clients that access it. To that end, we *permit the clients to vote for dynamic object replicas to be created at a specific site*. These votes are collected at each node and disseminated across the cluster so that the objects which received the most votes (or *popular* objects) are identified by the storage system. Dynamic replicas of such popular objects are created at sites where they are popular, and deleted when their popularity drops. When trying to access an object, clients tentatively determine the location of its closest replica (either static or dynamic) and address requests directly to the node holding it. Such approach however raises a number of challenges.

We acknowledge that using client votes has been proposed before in the context of replicated relational databases, specifically to ensure data consistency [36]. Transposing this idea to decentralized storage systems poses a number of significant challenges that we address in this paper. Specifically, collecting client votes efficiently without using a centralized process requires us to propose a novel, fully-decentralized, loosely-coupled vote collection algorithm. While existing dynamic replication techniques leverage a centralized repository to direct client requests to the nearest available replica, we propose a technique allowing clients to tentatively locate the closest available replica without any prior request to any of such repositories.

5.1. Collecting and counting votes

The goal of dynamic replication is to improve storage operation latency for the clients. Therefore, we need to design an efficient way to let the clients cast their votes for objects.

Collecting votes also raises a major challenge. While determining the most voted-for objects at each node is straightforward and can be done efficiently, inferring from this the most popular objects *cluster-wide* is not an easy task. This problem is named *distributed top-k monitoring*. Sadly, most implementations in the literature [37, 38, 39, 40] are centralized.

We address both these issues by mixing an approximate frequency estimation algorithm with the existing, lightweight Gossip protocol provided by storage system (Section 6).

5.2. Tunable replication

Determining when to create a new dynamic replica or delete an existing one based on the previous information is also challenging in a distributed setup. Indeed, it is necessary to bound the number of replicas to be created at each site without any node being responsible for coordinating the replicated items. Consequently, nodes must synchronize with each other before creating a new replica of any object. One could consider using a consensus protocol such as Paxos [41]. However, we argue it would be an overkill for such a simple task as Paxos is by no means a light protocol [42].

It turns out that the technique we use to solve the vote collection and counting challenge also provides all the information we need to solve this issue (Section 7).

5.3. Dynamic replica location

We need to enable clients to locate *dynamic replicas* as they do for *static replicas*. Obviously, such replicas should also be placed at a predictable, deterministically chosen node. We achieve this using the DHT-based data distribution of the storage system (Section 7.1). To access the closest available replica, a client also needs to know whether or not a dynamic replica exists at a given site. Systematically probing nearby sites for available replicas would contradict the single-hop read feature of Dynamo. Also, this would significantly increase storage operation latency, consequently missing the point of dynamic replication which is precisely to reduce this latency.

We demonstrate that this issue can be solved using probabilistic algorithms (Section 8). Our proposal builds on the solutions we adopt for the two aforedescribed challenges.

6. Identifying hot objects with client votes

In this section, we describe how to identify the most popular objects at each site. This is achieved in three steps:

1. We describe an efficient way to **allow the client to vote for an object to be replicated dynamically** at a specific location (Section 6.1).
2. We maintain a local **count of these votes at each node** to identify the most voted-for objects (Section 6.2).
3. We **disseminate and merge these votes** throughout the cluster to provide each node with a vision of the most popular objects for each site, cluster-wide (Section 6.3).

The identification method we describe in this section addresses the vote collection and counting challenge above.

6.1. Client vote casting

Clients vote for objects to be replicated dynamically at sites *close* to them. We name these sites *preferred sites*. This proximity can for instance express network latency, but also metrics such as bandwidth cost or available computational power may also be considered. To this purpose, each client maintains a list of such preferred sites, ordered by preference.

We argue that existing read queries to the storage system provide an ideal base for vote casting, as clients intuitively vote only for objects they need to read. In contrast, objects being written-to are not good candidates for dynamic replication because of the synchronization needed to keep dynamic replicas in sync with static replicas; we discuss write handling in Section 7.3. For every storage operation on an object, the client indicates in the request message its preferred sites for this object, *i.e.* the sites where the client would have preferred a dynamic replica of the object to exist. Let us assume a client wants to read the object associated with the key *key*. The client sends the request to the closest node n holding a replica of that object. Say this node belongs to a site s. We detail the location of this closest node in Section 8. The client piggybacks the request message with the list of the subset of sites having a higher preference than s in its list of preferred sites. Such request is interpreted by n as a vote for this object to be replicated on these sites.

6.2. Node-local vote collection and hot object identification

At each node, we want to know for each site the most voted-for objects. These are considered as *candidates* for dynamic replication. Each time a node receives a read request for an object identified by *key*, it records the vote for this object to be replicated on all sites indicated as preferred by the client. Let us first assume that we keep one counter per key and per site, which is incremented by 1 for each vote. We name *site counters* the set of key counters for a single site, and *vote summary* the set of site counters for all sites. In addition, if the object replica identified by *key* is a *dynamic* replica, we consider that the client *implicitly votes for this replica to be maintained*. As such, we also record the vote for *key* on the *local* site of the node receiving the request, *i.e.* the site the node belongs to. Algorithm 1 details these actions by a node receiving a read request from a client.

However, the goal of this scheme is to adapt to fluctuating object popularity by replicating dynamically the objects having the highest popularity over a *recent* period of time. Consequently, we use successive *voting rounds*. We extract at the end of each round the most voted-for objects for each site, and create a new, empty vote summary for the subsequent round. The length of a round is a cluster setting: we discuss its value in Section 11.2. We synchronize these rounds across the cluster by using the local clock of each node.

Keeping an exact vote summary for any given round is memory-intensive. It has a memory complexity of $O(M * S)$, M being the number of objects voted-for in this round and S being the number of sites in the cluster. Such complexity is not tolerable as billions of objects may exist and be queried in the cluster. Luckily, we do not need to keep the vote count for *all* objects: we are only interested in knowing which are the *most* voted-for objects for each site. For each site, finding the k most frequent occurrences of a key in a stream of data (client votes) is a problem known as top-k counting. Multiple approximate, memory-efficient solutions to this problem exist in the literature. In the context of our system, such approximate approaches are tolerable as it is not critical to collect the exact vote count for each object as long as the estimation of their vote count is precise enough and the set of objects identified as popular accurately captures the votes expressed by the clients. As such, we use as vote summaries a set of approximate top-k estimators, k being a configuration setting whose value is discussed in Section 11.1. We choose to use the *Space-Saving* algorithm [43] as top-k estimator. It guarantees strict error bounds for approximate counts of votes, and only uses limited, configurable memory space. Its memory complexity is $O(k)$. For any given site, the output of Space-Saving is the approximate list of the k most voted-for keys, along with an estimation of the number of votes for each.

Any given node simultaneously maintains $|S|$ active structures, one for each node in the cluster. Each time this node receives a request for an object v, for each preferred site indicated in the request, the key of v is added to the corresponding active structure. Consequently, at any time, a node is able to know which are the most frequent replication preferences indicated by a client for any site over the previous time window.

6.3. Cluster-wide vote summary dissemination

In this section we explain how to obtain the most voted-for objects across all nodes, starting from local vote summaries built from user votes (1). We periodically share the local vote summaries of each node with

Algorithm 1 Node-local object vote counting

Input: *key*: key of an object to read, *prefs*: list of preferred sites provided by the client.

```
procedure CountClientVotes(key, prefs)

    ▷ Interpret reading a dynamic replica as an implicit vote
    let local be the local site of the current node
    let replica be the local replica of the object with key key
    if the replica is a dynamic replica then
        add local to prefs
    end if

    ▷ Add client votes to the local vote summary
    for each preferred site site in pref do
        let vs[site] be the site counter structure for site
        count one vote for key in vs[site]
    end for

end procedure
```

its peers (2). Merging these peer vote summaries (3) gives each node a view of the most popular items across the cluster. Figure 1 illustrates this process.

We organize the process at any given node n in successive phases. During a voting round r of duration t, the local vote summary capturing client votes is named *active* summary. When a round ends, the summary transitions to a *merging* state: the node sends this summary to its *peers*, *i.e.* every other node in the cluster. Rounds being synchronized across the cluster, the node also receives summaries from its peers for *the same voting round*, which are merged with the local summary; merging this local summary with another one received from a peer n' gives a summary of the votes received by both n and n'. When vote summaries for every peer have been received, the summary is *complete*, at which point *all votes received by all nodes in the cluster* for the round r are summarized. After a period $2 * t$ since the round started, this cluster-wide summary transitions to a *serving* state which we detail in Section 7. We illustrate these successive vote summary phases in Figure 2.

In presence of faults, a summary can reach the *serving* state without having received all peer vote summaries in time. This may occur in case of delayed or lost packets. We qualify such summary as *incomplete*.

Such an approach is consistent with the class of systems we target: Dynamo provides an efficient algorithm for disseminating information across the cluster: *Gossip*. We use it to share a vote summary with every other node when it reaches the *merging* state. This approach is also compatible with our design choices: the *Space-Saving* structure we use is proven to be *mergeable* in [44], with a *commutative* merge operation.

Formally, we name MergeCounters(a, b) the function outlined in [44] that merges two Space-Saving structures a and b. MergeSummaries(v, v') is the function merging two vote summaries s and s'. These summaries contain site counters, respectively $v_1, ..., v_S$ and $v'_1, ..., v'_S$. S is the total number of sites in the cluster. This function returns a merged summary v'' containing S site counters $v''_1, ..., v''_S$, such that:

$$\forall a \in [1, S], vc''_a = \text{MergeCounters}(vc_a, vc'_a) \quad (1)$$

Considering that MergeCounters is commutative, it is trivial that MergeSummaries has the same property.

Let us assume a reliable network at this point, with all peer summaries being received *before* the local summary reaches a *serving* state. Because each node sends to all its peers *the same* local vote summary, and because the MergeSummaries function is commutative, the resulting *complete* summary after all peer summaries are merged is identical at each node. When all nodes reach the *complete* summary state, they share *the same view of the most voted-for objects* for each site. We use it to perform dynamic object replication in Section 7. We discuss the memory complexity of the popular object identification process in Section 11.4.

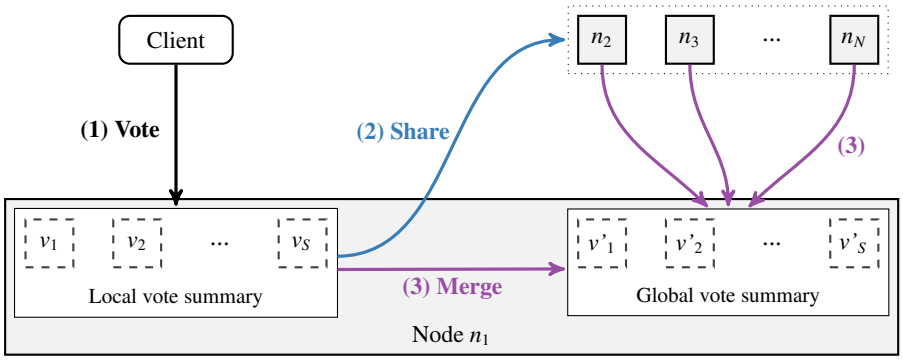


Figure 1: Cluster-wide popular object identification overview. S is the total number of sites in the cluster and N the total number of nodes.

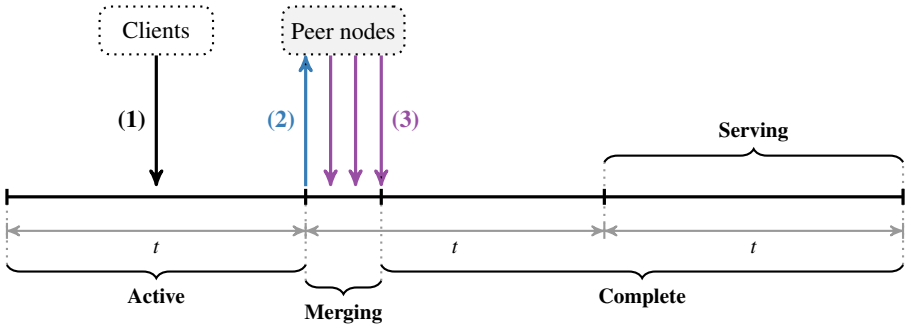


Figure 2: Timeline of vote summary states for a voting round length t.

7. Lifecycle of a dynamic replica

We detail in this section how to create and delete dynamic replicas using the cluster-wide vote summaries while handling writes to dynamically-replicated objects.

1. We first explain **when to create a new dynamic replica** of an object identified as popular on a site (Section 7.1).
2. We describe the process of **removing those data replicas** when their popularity popularity drops (Section 7.2).
3. We finally explain the process of forwarding writes to these dynamic replicas while retaining underlying storage system consistency and guarantees (Section 7.3).

7.1. When and where to create a dynamic replica?

With access to a shared vote summary, deciding when to create a replica is straightforward. Nodes in the cluster create remote dynamic replicas of the popular objects they are *primary node* for. As soon as a vote summary reaches the *complete* state, thus summarizing the votes of all clients across the cluster, the top-k most popular objects are replicated to sites at which they are popular. To replicate an object *obj* identified by a key *key* on a remote site s, a node n first informs all nodes holding *static* replicas of *obj*, which store this in their local state. Upon acknowledgement from these static replica nodes, the primary node updates its local state as well, and copies *obj* to a node at site s. The node on which this replica is placed is selected deterministically. In the case of Dynamo, we can use the existing DHT to place objects at a site in such a deterministic fashion. Starting on the cluster ring from the position $\text{H}(key)$, we walk the ring until we find a node at s, on which the replica is placed. Such a method is used today by Cassandra [30] for rack-aware data placement. Thus, assuming that a client knows a dynamic replica of an object exists at a site, it can easily infer which node holds this replica and address its request directly to it.

Fault-tolerance: *No replicas are created based on incomplete vote summaries*. In the presence of failures, this may result in dynamic replicas of yet popular objects not being created at the initiative of its primary node. We handle this case with the replica read process we outline in Section 8.

7.2. When to delete a dynamic replica?

Each node is responsible for the deletion of any dynamic replicas it holds. A dynamic replica at a site s can be deleted if it is not among the top-k items for this site s in the *serving* summary for the current time period. We also want to avoid replica *bounces*, *i.e.* object replicas being repeatedly created and deleted at the same site. This may happen for objects whose popularity ranking is around the top-k threshold, and fluctuates above and under this threshold. We define a grace period g, which represents the minimum number of *consecutive* serving summaries a previously-popular object must be absent from before its dynamic replica is deleted.

The deletion process is simple. To delete a locally-held dynamic replica of an object *obj*, a node flags it as *inactive*: subsequent client requests for that replica are as if it does not exist, which we discuss in Section 8. The node then informs the static replica nodes of *obj* of the deletion of this replica, which they remove from their local state.

Fault-tolerance: *No replicas are deleted based on incomplete vote summaries*. Replicas of objects whose popularity dropped may not be deleted as they should in the presence of failures. Yet, this principle ensures that no replicas of still popular objects are ever deleted, and remain accessible.

7.3. Handling writes to dynamically replicated objects

Our dynamic replication scheme enables the storage system to handle writes to dynamically-replicated objects. The afore-detailed dynamic replica creation makes this process straightforward. Clients address write requests for any object to one of the static replica nodes of this object. Based on their local state, these nodes determine all existing dynamic replicas of that object and propagate the write to all other replicas, static *and* dynamic, using the write protocol of the storage system.

Correctness: because our replication system does not modify the write propagation algorithm of the underlying system, writes are propagated to dynamic replicas the same way they are propagated to static replicas. In Voldemort, writes are eventually-consistent by default, and can optionally be made strongly consistent. Our proposal shows the same consistency characteristics. As such, we only need to ensure that the creation and deletion of dynamic does not violate these consistency characteristics. We ensure this by persisting the new dynamic replica list for each object on its origin nodes before creating and after deleting any dynamic replica. This guarantees that writes to dynamically replicated objects are always forwarded to all of its dynamic replicas.

We prove that this write protocol is correct, *i.e.* does not cause dynamic replicas to be out of sync with static replicas, even in the case of system failures. A dynamic replica is created only *after* successful acknowledgement from all other static replica nodes. These nodes only are informed of the replica deletion *after* it is flagged as inactive. This ensures that writes to an object are always propagated to the dynamic replicas of that object, even in the presence of faults.

8. Accessing the closest replica

In this section, we explain how to let clients access a close replica of objects they want to read, either static or dynamic, without any communication with any dedicated metadata node. In a nutshell, we let the user request information about dynamic replicas created on its preferred sites at any given time, and later use this data to infer the location of the closest replica of any object and access it directly.

8.1. Locating the closest replica: dynamic replica summaries

We assume a client can know at any time the list of all active dynamic replicas in its preferred sites. This assumption greatly eases locating the closest replica: it is the static or dynamic one located on the site with its highest preference, or the closest static replica if no replica exists at any preferred site. We name this list of dynamic replicas for a site *s dynamic replica summary* of s, and detail the closest replica location in Algorithm 2. The client addresses its request to the node holding this replica on the site indicated by this algorithm using its knowledge of the cluster DHT. The node receiving this request returns the dynamic replica, if available. If it is not available locally, it forwards the request to the closest static replica node, which is guaranteed to hold the object.

We near these assumptions by enabling any client to request from *any* node such dynamic replica summary at its preferred sites at the time of the request. Any node is able to answer this request based on the cluster-wide vote summary dissemination process. Using its knowledge of the client votes across the cluster given by the current *complete, serving* summary, the node knows the list of all dynamic replicas currently active at any site. This list is sent to the client for its preferred sites.

Algorithm 2 Closest replica site inference

Input: *key*: key of an object to read, S: preferred sites list

```
function InferClosestReplicaSite(key)
    for each site s in S do
        let R[s] be the dynamic replica summary for s
        if s is a static site for key or key in R[s] then
            return s
        end if
    end for
    return the closest static replica of val
end function
```

Luckily, the client request routing provided by Dynamo enables us to only *infer* the location of the closest available replica without jeopardising the read protocol correctness. In case of inference error, the client will address its request to a wrong node, but this node will forward it to a static replica node which is guaranteed to hold the object. This enables us to use an approximate structure as a memory-efficient way to represent this list of dynamic replicas: Bloom filters [45].

8.2. When to refresh dynamic replica summaries?

Intuitively, clients need to periodically refresh dynamic replica summaries to account dynamic replicas being created and deleted. We provide a simple yet efficient way to let a client decide of the appropriate time to do so: *error indication*.

Two different types of errors can be caused by our closest replica location algorithm: *false positives* – when a node is wrongly believed to hold a replica of the object, and *false negatives* – when the request is sent to a non-optimal replica of the object, *i.e.* when a closer replica existed according to the preferred sites of the client. A false positive happens if the replica summary is outdated and the replica was deleted since its last update, or in case of a false positive caused by the Bloom filter. A false negative occurs if the replica summary was last updated *before* this replica was created.

Nodes in the server are able to indicate to clients such false positives (if the node being sent a read request does not hold the requested dynamic replica) or false negatives (based on the latest serving vote summary they hold) by flagging the response to a request accordingly. Clients keep a local counter of such inference errors and decide to refresh their dynamic replica summaries and reset this counter when it reaches a configured threshold. We discuss its configuration in Section 11.3.

To cope with cases where the configured summary validity period is too large for some usage patterns, the client keeps a count of false positives or false negatives. Should the client address the request to a server not holding the desired piece of data, the server relays the query as usual to the correct node but flags the answer as erroneous. This may happen for instance because it has been deleted after the client bloom filter has been last updated. We call this case a *false positive*. Inversely, if the client accesses a remote object replica and a closer one existed, which we call a *false negative*, the server responding to the client flags the response message as well. A counter of wrong assumptions is maintained by each client, and set to 0 every time the summary is updated. For each false positive or negative, the client increments its counter. It updates its bloom filter as soon as the counter reaches a configurable threshold.

9. Prototype implementation

We implement our approach atop a real-world key-value store. Voldemort [14] is an open-source clone of Dynamo [13]. It is developed by LinkedIn and is extensively used in their software stack [46]. Voldemort is a modular Java application, making it a relevant choice for our experiments. The total of our additions account for about 2,800 lines of Java code, excluding open-source Space-Saving and Bloom Filter libraries. We modify both the server code to handle decentralized object replication (Section 9.1) and the client code to perform close replica location (Section 9.2).

9.1. Server-side modifications

Our server-side modifications are designed as a middleware. We extensively use the native functionality provided by Voldemort to implement our prototype: Gossip and local persistent storage interface. Incoming client requests are first processed by this middleware to extract site preference information that is used

to update the local popularity measurement structure (Section 6.2). That structure is disseminated using the Voldemort gossip protocol (Section 6.3). A background thread is responsible for maintaining the cluster-wide popularity measurement structures using the information received from other nodes. This information is used to create or delete remote, dynamic object replicas. Such replicas are stored locally using Voldemort persistent storage.

The most tricky and challenging part of the implementation concerns write operations. We have to substantially modify write handling so that a node holding a dynamic replica of a data object behaves as if it held a static replica of that object, forwarding the request to all other nodes holding replicas of that object according to its local state as detailed in Section 7.3.

9.2. Client-side changes

Modifications on the client were kept to the minimum. Each client maintains a statically-configured list of site preferences. Each storage operation request for any given object is routed according to our replica location algorithm outlined in Section 8.1. If no dynamic replica for this object can be found in the local Bloom filter, the request is routed according to Voldemort vanilla algorithm. Site preferences are piggybacked to the request. Incoming responses are checked for *false positive* or *false negative* flags using a specifically designed middleware in order to update the local error counter described in Section 8.2.

10. Experimental evaluation

In this section we prove the effectiveness of our approach using our prototype. We perform this in 4 steps:

1. We first show that our object heat measurement technique presented in Section 6 is able to properly identify the hottest objects in the cluster (Section 10.1).
2. We demonstrate that the object replication method described in Section 7 effectively replicates these hottest objects (Section 10.2).
3. We prove that our object location technique introduced in Section 8 is accurate (Section 10.3).
4. We finally confirm that the combination of these principles applied to Voldemort efficiently reduces the average read latency under different workloads when compared to other caching-based approaches (Section 10.4).

Experimental platform. We deploy our prototype on 96 nodes of the Amazon EC2 cloud. For all our experiments, we use *t2.large* general-purpose instances, evenly distributed over 6 sites spanning 4 continents: California, Virginia, Ireland, Germany, Australia, and Japan. Each virtual machine has access to 2 CPU cores and 8 GB RAM. The host server is outfitted with 10 Gigabit ethernet connectivity. Our measurements show the main bottleneck of these virtual machines to be the CPU. Single-site experiments on bare-metal servers show higher throughput than Amazon EC2.

Dataset and workload. The main workloads we target are dominated by reads, such as the applications described in the motivation section above. Yet, we prove that even for write-intensive workloads, our data replication system increases average read performance at the cost of a slightly increased write latency due to the added cost for dynamic replica synchronization. To that extent, we use YCSB [47] to generate both our initial dataset and storage operations. YCSB is an industry-standard benchmark commonly used to evaluate the performance of key-value stores. We generate 500 million 1 KB records, that we insert in Voldemort as our initial data. The workload against the cluster is generated by two YCSB instances on each site (12 in total) running in their own virtual machine. Requests are generated according to a Zipfian distribution, with Zipf parameter $\rho = 0.99$, drawing from a set of 20 million keys. Each measurement is performed 50 times at maximum throughput for a period of 5 minutes. Bar plots represent averages. To account for changing hot object set, YCSB instances are restarted in turn – one every 30 seconds. This leads to a varying request distribution over time. We collect all measurements on a stable, non-saturated cluster to ensure that our measurements do not suffer from compute resource exhaustion on the server. Requests are locally throttled at each node to a maximum of 1,000 concurrently running queries.

Experimental configuration. Our prototype uses the default time window length of 10 seconds, a grace period of 3, a k value of 10,000, and a client error threshold of 5%. Client-side site preferences are identical for every node in the same site. In the applications we target, both database clients and servers are located in a well-defined datacenter. Consequently, the preference list for each client

Table 1: Experimental client preferred site settings

Site	1st preference	2nd preference	3rd preference
California *(us-west-1)*	California *(0.8 ms)*	Virginia *(72.8 ms)*	-
Virginia *(us-east-1)*	Virginia *(0.8 ms)*	California *(72.8 ms)*	Ireland *(81.2 ms)*
Ireland *(eu-east-1)*	Ireland *(0.7 ms)*	Germany *(21.4 ms)*	Virginia *(81.2 ms)*
Germany *(eu-central-1)*	Germany *(0.8 ms)*	Ireland *(21.4 ms)*	-
Australia *(ap-southeast-2)*	Australia *(0.9 ms)*	-	-
Japan *(ap-northeast-1)*	Japan *(0.7 ms)*	-	-

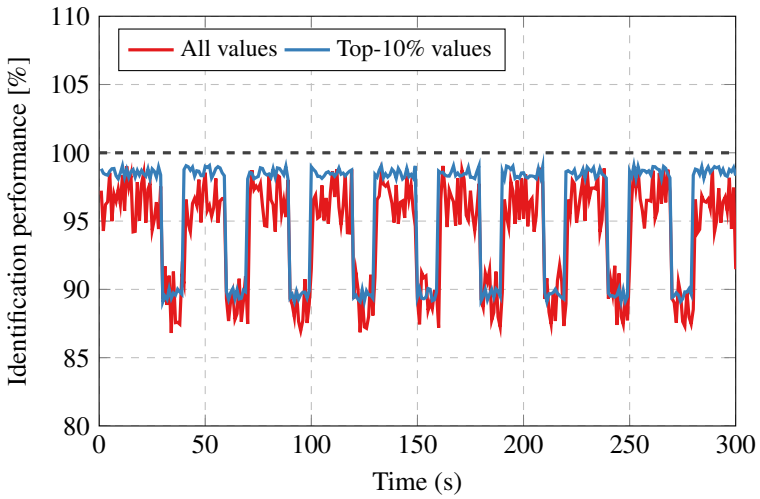


Figure 3: Popular object identification performance.

can be statically configured – showing no significant client overhead. Preferred sites are ordered by average measured round-trip latency, up to a maximum of 100 ms. They are configured as indicated in Table 1 – each site being its own first preference. We keep Voldemort default configuration settings.

Evaluated cache algorithms. We compare our approach to cache-based algorithms: Least Recently Used (LRU), Most Recently Used (MRU), Random Replacement (RR), and Adaptive Replacement Cache (ARC) [48]. We implemented these algorithms in Voldemort, so that each clients contacts the local server that would hold the requested piece of data (as explained in Section 8) if cached.

10.1. Popular objects identification

We show in this section the behavior of our popularity measurement scheme introduced in Section 6. We run a read-only workload, as popular objects are only identified based on client read queries. We log for each YCSB instance the object keys requested. Collecting this information from every request generator enables us to calculate the exact set of k most frequent popular objects for each site across the cluster at any given time. We also log on each node the cluster-wide popularity summaries, at a one-second interval. Combining those logs with the aforementioned request logs enables us to derive a key metric: *identification performance*. Identification performance corresponds to the proportion of identified popular objects at any given time compared to our exact, offline computation. A popular object is considered identified if it is present in the cluster-wide object heat summaries of *all* nodes of the cluster. This proportion is measured in percents, 100% indicating a perfect match between theoretical and practical results.

We plot our results in Figure 3. Overall, they show that more than 95% of the hot couples are identified by our heat measurement and dissemination scheme. We can observe a regular identification performance drop of about 8%, repeated every 30 seconds. Each of these drops corresponds to one request generator being restarted, leading to a sudden change of the popular keys being requested by that generator. For each of these events, we note that the cluster convergence time

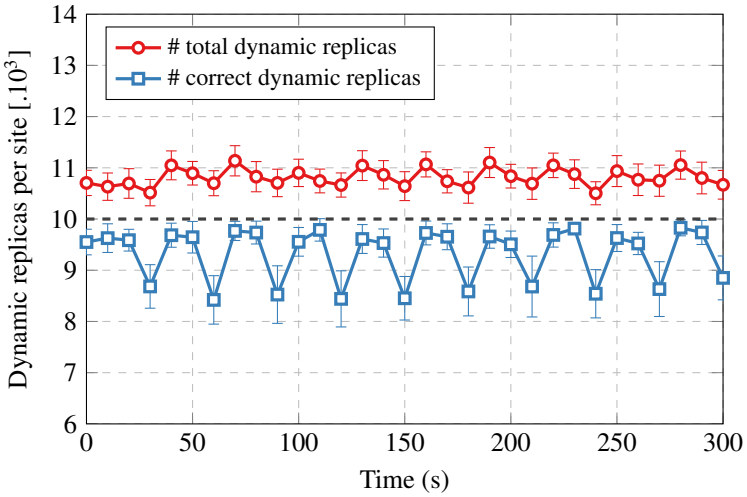


Figure 4: Popular object replication performance.

stays in the order of 12 seconds. This specific time is explained by our popularity measurement time window configured to 10 seconds, plus the Gossip protocol convergence time for the popularity measurements dissemination cluster-wide, which we measured to take about two seconds.

We note clear variability in our measurements. This is caused by the rapid fluctuation of popularity computations near the limit of the top-k estimator we use in our popular data identification scheme. We made sure of this by plotting the identification performance for the 10% most popular objects. We observe that this blue curve is much more stable than the curve for all values, meaning that the most popular objects requested by clients are identified accurately.

We ran the same experiments with object sizes of 100KB, 1 MB, 10 MB. The results is a replication time increase of respectively 1.2%, 3.8% and 6.2% compared to 1 KB size, due to the added network bandwidth required for the data transfers.

It is interesting to note that our measurements show that our hot object identification scheme never reaches 100% accuracy. This is because of the unstable frequency estimations for keys near the detection limit of the Space-Saving structure, which changes significantly faster than the convergence time of the cluster. This also explains the noticeable variability of our measurements.

10.2. Dynamic replica creation and deletion

In this experiment we show the replication performance achieved by the principles we introduced in Section 7. As for popular object identification, only read operations are considered for replication: we run a read-only workload. We execute it for a duration of 5 minutes and calculate at each site the number of replicated objects. We plot the results we obtain in Figure 4, with 95th percentile confidence intervals. The red curve indicates the average number of active replicas per site at any given time. The blue curve indicates, among all created dynamic replicas, the average number of replicas which are correct, *i.e.* for which the object is present in the offline-computed, theoretical top-k for the corresponding site calculated as in the previous experiment.

We first focus on the correct replica count curve, in blue. The regular drops are caused by YCSB clients being restarted at regular intervals, which impact the distribution of the key requests as previously observed. We note that this drop is only temporary, and that the correctness of the replicas returns to normal quickly, after new replicas are being created, within approximately 12 seconds as in the previous experiment. We show that our replication technique replicates a little more than 95% of the actual top-k most popular objects, which is coherent with the popular object identification performance we could observe in the previous experiment.

We note on the red curve that the count of active dynamic replicas is relatively stable at about 10,600 per site on a stabilized state. This is slightly more than the configured value of k (10,000) because of the grace period of 3, which causes replicas to be deleted with a slight delay when their popularity drops. We also notice that this curve stays at all times *above* the configured k, because the top-k items as identified by the nodes are always replicated. The number of active repli-

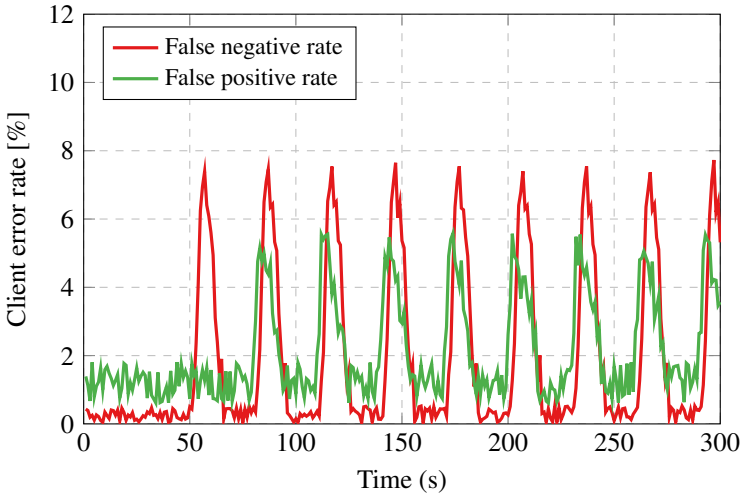


Figure 5: Client location error rates.

cas increases slightly after each YCSB client restart because of the grace period, which causes a number of new replicas being created without deleting immediately the previously-popular objects. Once the grace period has elapsed, these dynamic replicas are removed, which causes the total number of replicas to decrease after these spikes.

10.3. Replica location

We show the accuracy of our replica location scheme (Section 8) using a read-only workload. We measure the afore-mentioned *false positive* and *false negative* rates at the clients. We plot the results in Figure 5.

Let us focus first on the false negative curve, drawn in red. The spikes observed at regular intervals are caused by the restart of YCSB clients, which changes the distribution of the requested keys consequently causing new replicas to be created. Clients are only informed of these new replicas with error indication from the server nodes, explaining the rapid increase in false negative rate. As soon as this rate crosses the set threshold (5%), the clients update their local summary, causing the number of false positives to decrease rapidly. The first spike occurs at about 50 seconds, while the first client restart occurs at 30 seconds. This is because of the two 10-second rounds required for a summary to reach a *serving* state.

The false positive curve, plotted in green, shows the same patterns. Spikes are caused by dynamic replicas being deleted from the cluster, after the end of the grace period. As with false negatives, error indication enables clients to detect these errors and update their summaries within less than 10 seconds.

10.4. Evaluating the whole strategy: latency impact

In this section we evaluate the latency impact of our solution. To this end, we use two YCSB workload patterns to evaluate our scheme: read-heavy (95% reads – 5% writes) and update-heavy (50% reads – 50% writes). Due to the lack of dynamic replica placement systems to compare with, we choose to implement cache-based algorithms at all sites: Least Recently Used (LRU), Most Recently Used (MRU), and Adaptive Replacement Cache (ARC) [48]. Such edge-caching is one commonly-used solution to improve read latency of globally-distributed storage systems, as, for example, at Facebook [19]. We keep unmodified Voldemort as a baseline. We implemented these algorithms in Voldemort, so that each clients contacts the local server that would hold the requested piece of data (as detailed in Section 7.1) if cached. We do not implement writes for these caching algorithms, as they are not designed for that purpose. Using the original Voldemort algorithm we simulate writes to different keys than the ones being read.

We plot in Figure 6 the measured read latency of all five systems under a sustained read-heavy workload over time, with 95th percentile confidence intervals. It can be observed that the read latency achieved without caching or with MRU caching is stable, and is not affected by the changes in the distribution of requested keys caused by YCSB client restarts. We note that MRU causes higher latency than no caching at all. This is explained by our workload, which is the worst possible case for that algorithm: the most recent keys are the more likely to be requested. Yet, such caching prevents

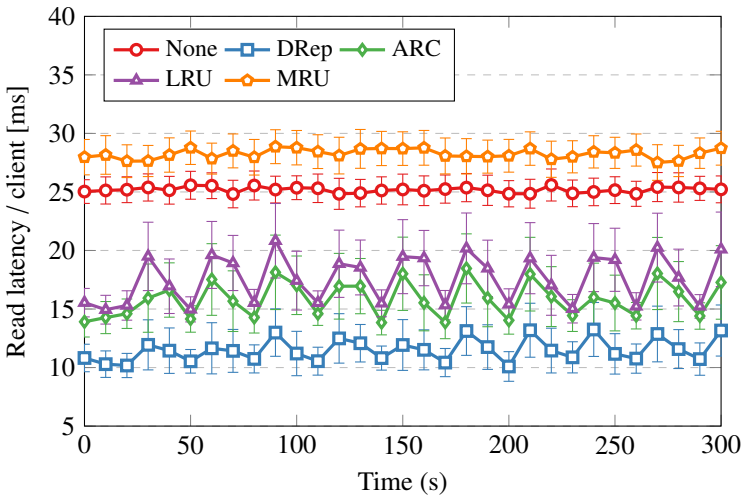


Figure 6: Latency over time with a 95% read / 5% write workload.

single-hop reads. Our dynamic replication scheme performs globally better than both ARC and LRU, showing lower latency spikes at YCSB client restarts. This is because we enable locating dynamic replicas *at close sites* while caching only allows site-local caching. The similarity between LRU and ARC curves is related to the fact that the latter is built atop the former.

Figure 7 shows the impact of the workload on the latency of respectively reads and writes of all systems. We implemented writes on ARC, LRU and MRU using asynchronous cache eviction technique as implemented at Facebook [19]. As a write is received on any given key, an asynchronous event is triggered and broadcasted to every datacenter in order to evict this key, if cached, from the local cache.

On Figure 7a, we show that read latencies tend to increase as the ratio of reads over writes decreases, but keep the same relative order. We observe that the read performance advantage of our method, previously observed with a 95% / 5% read-to-write ratio is preserved even when this ratio decreases. We also note that unmodified Voldemort and our dynamic replication method show only marginally increasing latencies as the read-to-write ratio decreases. This is because of the relatively low overhead of the write protocol, which only broadcasts the write notifications to datacenters effectively holding a replica of the keys being written to.

Figures 7b details this last point, showing the write latency of our dynamic replication method being slightly higher than the one of unmodified Voldemort. This is due to the additional bookkeeping required to keep dynamic data replicas synchronized with static replicas. The latency showed by other caching methods is comparable, as the write protocol is the same for each method. However, the overhead of broadcasting write events to every datacenter significantly hinders the write latency.

This demonstrates that although we targeted our geo-replication method to read-intensive workloads with rare writes, the read latency is also significantly decreased compared to unmodified Voldemort and edge caching-based approaches for balanced and even write-intensive workloads, at the cost of a sensible write latency increase.

11. Discussion

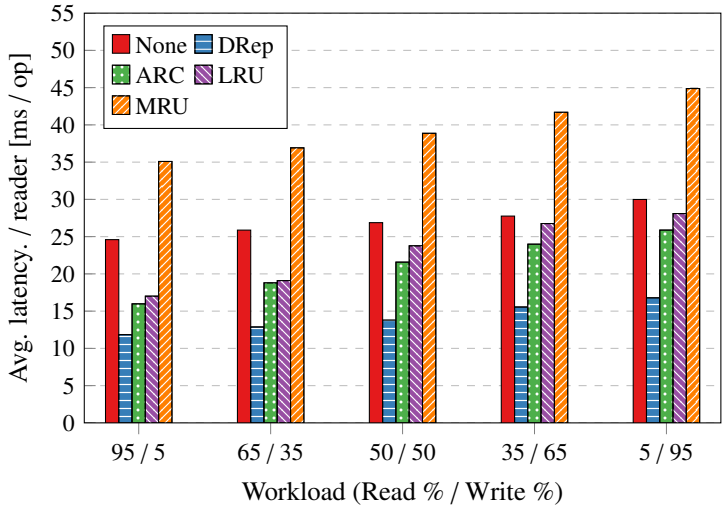
11.1. Bounding the number of replicated objects: choosing k

The number of replicated objects at a site s is strongly tied to the configuration of k for that site, that we name k_s. This is because all top-k_s voted-for objects are replicated dynamically at s. We deduce that, at any time, there are at least k_s elements replicated on site s. Dynamic replicas are deleted when their popularity drops under the top-k_s threshold. Assuming that λ such that $0 \leq \lambda \leq 1$ is the average replacement ratio of new elements in the top-k of any round compared to the previous one, the number R_s of dynamic replicas available at site s is:

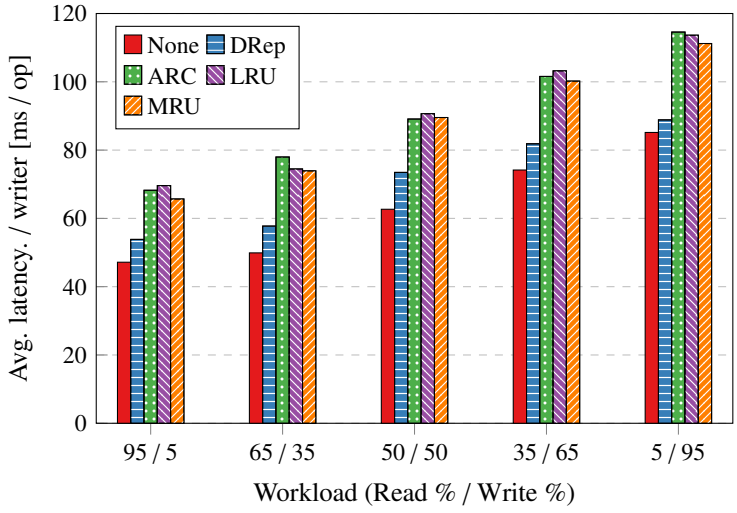
$$R_s = k_s + (k_s * \lambda g) \tag{2}$$

We extract from this equation the value of k_s for an available dynamic replication capacity C_s at site s such that $R_s \leq C_s$:

$$k_s \leq \frac{C_s}{\lambda g + 1} \tag{3}$$



(a) Workload read latency impact



(b) Workload write latency impact

Figure 7: Latency impact of the workload.

11.2. Configuration of the round length

The voting round length is an essential parameter for our replication scheme. It obviously impacts the object detection latency between the moment when an object is becoming popular at the clients and the time when it is identified as such by the cluster and replicated. The lower the round length, the faster the identification. However, two things are crucial to consider when choosing its value. First, a low value will cause vote summaries to be disseminated at more regular intervals between nodes, increasing the network overhead. But more importantly, the vote round length should be set to be *much higher than the Gossip convergence time of the cluster*. Failing to consider this parameter results in more frequent incomplete summaries at the nodes because of delayed network packets, and consequently in poor replication performance.

11.3. Keeping client errors low: setting the error threshold

The error threshold is set up as a ratio of errors to the number of requests processed. It has to be configured relatively to the performance requirements of the application accessing data. The lowest the expected latency, the lowest this ratio. In our experiments on replica location (Section 10.3), we show that a set value of 5% allows nodes to detect quickly new replicas while not updating replica summaries too often. Good balance between these two parameters is essential: setting an error threshold too low may be counter-productive because of the additional network overhead and computational power required by summary refreshes.

11.4. Memory complexity

Hot object identification algorithm The memory complexity of Space-Saving structures is configurable: $O(k)$. We deduce that the memory complexity of our popular object identification algorithm is $O(k * N * S)$. Evaluation on a 96-node cluster gives good results. We plan to further explore its worst case scenario: thousands of nodes, or more.

Location algorithm Bloom filters are a memory-efficient way to test element membership in a list. It has a configurable error, impacting its accuracy. We choose an error probability of 0.001. With $k = 10,000$, the size of a dynamic replica summary for a single site is under 18 KB.

11.5. Bounce period efficiency

In our experiments with a 95% / 5% read/write ratio, we observed that disabling the bouncing period (setting it to 1) increased network traffic by an average of 5%, because of some objects showing a popularity near the threshold to be unnecessarily, repeatedly created and deleted ; further increasing the bouncing period leads to no substantial network traffic reduction.

12. Conclusion

In this paper we present a decentralized dynamic replication strategy that integrates with the existing architecture of existing state-of-the-art storage systems, which enables us to leverage their existing, built-in algorithms to efficiently handle geo-distributed read and writes. We allow the clients to efficiently locate a close dynamic replica of the data without prior communication with a dedicated metadata server. We deploy a prototype implementation of these methods on 96 nodes distributed over 6 locations of the Amazon EC2 cloud. Experiments show a latency improvement of up to 42% compared to other state-of-the-art, caching-based solutions.

We are working on further enhancing our approach. This includes designing a write-aware replication scheme better able to handle balanced read/write workloads, improving the horizontal scalability of the popular object identification scheme, and generalizing it to a variety of different storage systems. Finally, we want to explore the behavior of our proposal with a wider range of parameters and workloads, and formalize the configuration trade-offs.

Acknowledgment

This work is part of the BigStorage project, supported by the European Commission under the Marie Skłodowska-Curie Actions (H2020-MSCA-ITN-2014-642963). The authors would like to thank the anonymous reviewers whose counsel and expertise greatly contributed to this research. The experiments presented in this paper were carried out on the Amazon Web Services infrastructure provided by Amazon through the AWS Cloud Credits for Research program.

References

[1] Microsoft Azure, `https://azure.microsoft.com/en-us/` (2016).

[2] Amazon Web Services, `https://aws.amazon.com/` (2016).

[3] K. Bogdanov, M. Peón-Quirós, G. Q. Maguire, Jr., D. Kostić, The nearest replica can be farther than you think, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, ACM, New York, NY, USA, 2015, pp. 16–29. doi:10.1145/2806777.2806939.

[4] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, C. Stratan, Monalisa: An agent based, dynamic service system to monitor, control and optimize distributed systems, Computer Physics Communications 180 (12) (2009) 2472 – 2498, 40 {YEARS} {OF} CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures. doi:http://dx.doi.org/10.1016/j.cpc.2009.08.003.

[5] K. Aamodt, et al., The ALICE experiment at the CERN LHC, JINST 3 (2008) S08002. doi:10.1088/1748-0221/3/08/S08002.

[6] S. Acharya, S. B. Zdonik, An Efficient Scheme for Dynamic Data Replication, Tech. rep., Brown University, Providence, RI, USA (1993).

[7] R. K. Grace, R. Manimegalai, Dynamic replica placement and selection strategies in data grids— a comprehensive survey, Journal of Parallel and Distributed Computing 74 (2) (2014) 2099 – 2108. doi:10.1016/j.jpdc.2013.10.009.

[8] Y. Chen, R. H. Katz, J. D. Kubiatowicz, Dynamic Replica Placement for Scalable Content Delivery, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 306–318.

[9] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google File System, in: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, ACM, New York, NY, USA, 2003, pp. 29–43.

[10] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop Distributed File System, in: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–10.

[11] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, E. Harris, Scarlett: Coping with skewed content popularity in mapreduce clusters, in: Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, ACM, New York, NY, USA, 2011, pp. 287–300. doi:10.1145/1966445.1966472.

[12] D. S. Jayalakshmi, T. P. Rashmi Ranjana, S. Ramaswamy, Dynamic Data Replication Across Geo-Distributed Cloud Data Centres, Springer International Publishing, Cham, 2016, pp. 182–187. doi:10.1007/978-3-319-28034-9_24.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, SIGOPS Oper. Syst. Rev. 41 (6) (2007) 205–220. doi:10.1145/1323293.1294281.

[14] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, S. Shah, Serving large-scale batch computed data with project voldemort, in: Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 18–18.

[15] P. Matri, A. Costan, G. Antoniu, J. Montes, M. S. Pérez, Towards efficient location and placement of dynamic replicas for geo-distributed data stores, in: Proceedings of the ACM 7th Workshop on Scientific Cloud Computing, ScienceCloud '16, ACM, New York, NY, USA, 2016, pp. 3–9. doi:10.1145/2913712.2913715.

[16] J. Titcomb, Facebook approaches 2 billion users, http://www.telegraph.co.uk/technology/2017/05/03/facebook-approaches-2-billion-users/ (2017).

[17] C. Smith, Facebook users are uploading 350 million new photos each day, http://www.businessinsider.com/facebook-350-million-photos-each-day-2013-9 (2013).

[18] G. Carey-Simos, How much data is generated every minute on social media?, http://wersm.com/how-much-data-is-generated-every-minute-on-social-media/ (2015).

[19] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, V. Venkataramani, Scaling memcache at facebook, in: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), USENIX, Lombard, IL, 2013, pp. 385–398.

[20] M. Mihaljevic, Facebbok cache invalidation pipeline, https://www.usenix.org/conference/srecon15europe/program/presentation/mihaljevic (2015).

[21] X. Dong, J. Li, Z. Wu, D. Zhang, J. Xu, On dynamic replication strategies in data service grids, in: Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on, 2008, pp. 155–161. doi:10.1109/ISORC.2008.66.

[22] Q. Wei, B. Veeravalli, Z. Li, Dynamic replication management for object-based storage system, in: 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage, 2010, pp. 412–419. doi:10.1109/NAS.2010.24.

[23] H. Shen, Ead: An efficient and adaptive decentralized file replication algorithm in P2P file sharing systems, in: Peer-to-Peer Computing , 2008. Eighth International Conference on, 2008, pp. 99–108. doi:10.1109/P2P.2008.37.

[24] J. MacCormick, N. Murphy, V. Ramasubramanian, U. Wieder, J. Yang, L. Zhou, Kinesis: A New Approach to Replica Placement in Distributed Storage Systems, Transactions on Storage.

[25] M. K. Madi, Y. Yusof, S. Hassan, Replica placement strategy for data grid environment, Int. J. Grid High Perform. Comput. 5 (1) (2013) 70–81. doi:jghpc.2013010105.

[26] Akamai – Dynamic Page Caching, https://blogs.akamai.com/2015/10/dynamic-page-caching-beyond-static-content.html (2016).

[27] N. Mansouri, G. H. Dastghaibyfard, A dynamic replica management strategy in data grid, Journal of Network and Computer Applications 35 (4) (2012) 1297 – 1303, intelligent Algorithms for Data-Centric Sensor Networks. doi:http://dx.doi.org/10.1016/j.jnca.2012.01.014.

[28] K. A. Kumar, A. Deshpande, S. Khuller, Data placement and replica selection for improving co-location in distributed environments, CoRR abs/1302.4168.

[29] L. Suresh, M. Canini, S. Schmid, A. Feldmann, C3: Cutting tail latency in cloud data stores via adaptive replica selection, in: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI'15, USENIX Association, Berkeley, CA, USA, 2015, pp. 513–527.

[30] A. Lakshman, P. Malik, Cassandra: A decentralized structured storage system, SIGOPS Oper. Syst. Rev. 44 (2) (2010) 35–40. doi:10.1145/1773912.1773922.

[31] Riak KV, http://basho.com/products/riak-kv/ (2016).

[32] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, in: Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, STOC '97, ACM, New York, NY, USA, 1997, pp. 654–663. doi:10.1145/258533.258660.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '01, ACM, New York, NY, USA, 2001, pp. 149–160. doi:10.1145/383059.383071.

[34] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic algorithms for replicated database maintenance, in: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '87, ACM, New York, NY, USA, 1987, pp. 1–12. doi:10.1145/41840.41841.

[35] N. Hayashibara, X. Defago, R. Yared, T. Katayama, The φ accrual failure detector, in: Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on, 2004, pp. 66–78. doi:10.1109/RELDIS.2004.1353004.

[36] D. K. Gifford, Weighted voting for replicated data, in: Proceedings of the Seventh ACM Symposium on Operating Systems Principles, SOSP '79, ACM, New York, NY, USA, 1979, pp. 150–162. doi:10.1145/800215.806583.

[37] B. Babcock, C. Olston, Distributed top-k monitoring, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, California, 2003, pp. 28–39.

[38] M. Wu, J. Xu, X. Tang, W.-C. Lee, Top-k monitoring in wireless sensor networks, IEEE Trans. on Knowl. and Data Eng. 19 (7) (2007) 962–976. doi:10.1109/TKDE.2007.1038.

[39] S. Michel, P. Triantafillou, G. Weikum, KLEE: A framework for distributed top-k query algorithms, in: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, VLDB Endowment, 2005, pp. 637–648.

[40] P. Cao, Z. Wang, Efficient top-k query calculation in distributed networks, in: Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing, PODC '04, ACM, New York, NY, USA, 2004, pp. 206–215. doi:10.1145/1011767.1011798.

[41] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234. doi:10.1145/322186.322188.

[42] R. Van Renesse, D. Altinbuken, Paxos made moderately complex, ACM Comput. Surv. 47 (3) (2015) 42:1–42:36. doi:10.1145/2673577.

[43] A. Metwally, D. Agrawal, A. El Abbadi, Efficient computation of frequent and top-k elements in data streams, in: T. Eiter, L. Libkin (Eds.), Database Theory - ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 398–412. doi:10.1007/978-3-540-30570-5_27.

[44] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, K. Yi, Mergeable summaries (2012) 23–34doi:10.1145/2213556.2213562.

[45] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426. doi:10.1145/362686.362692.

[46] Voldemort at LinkedIn, https://engineering.linkedin.com/\architecture/brief-history-scaling-linkedin (2015).

[47] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with ycsb, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, ACM, New York, NY, USA, 2010, pp. 143–154. doi:10.1145/1807128.1807152.

[48] N. Megiddo, D. S. Modha, One Up On LRU, The Magazine of the USENIX Association 28 (4) (2003) 7–11.