Geo-Replicated Buckets

An Optrimistic Geo-Replicated Shim
for Key-Value Store

João Miguel Louro Neto

Master of Science Thesis
Stockholm, Sweden 2015

TRITA-ICT-EX-2015:226

KTH Computer Science
and Communication

Geo-Replicated Buckets

An Optimistic Geo-Replication Shim for Key-Value Stores

JOÃO MIGUEL LOURO NETO

Master's Thesis

Supervisor: Vianney Rancurel (Scality)
Supervisor: Marc Shapiro (INRIA, LIP6)
Examiner: Johan Montelius (KTH)

2015-11-12

TRITA-ICT-EX-2015:226

Abstract

This work introduces GeoD and VersionD. GeoD is a causally-consistent geo-replication shim for key-value stores. GeoD enables the separation of concerns regarding replication and convergence from key-value stores, while preserving all of their read-only functionality. VersionD extends the feature set of GeoD by providing a versioning API, allowing applications to store multiple values for a given object, and to use context-specific conflict-resolution rules. We also discuss an efficient architecture for optimistically replicated systems, where all the storage replicas persist data in a common shared DHT. Since most key-value stores don't support versioning and GeoD relies on versioning support from the key-value store, we developed a shim layer that also augments the traditional core API with extra versioning operations and logic.

Contents

1 Introduction **1**

2 Background **3**

2.1 Replication . . . 3

2.1.1 Objects, Replicas and Sites . . . 3

2.1.2 Geo-Replication . . . 3

2.1.3 Optimistic Replication . . . 4

2.2 Time and Ordering of Events . . . 4

2.3 Consistency and Conflicts . . . 4

2.3.1 Strong Consistency . . . 5

2.3.2 Weak Consistency . . . 5

2.3.3 Hybrid Consistency Models . . . 6

2.4 Distributed Hash Table . . . 6

2.5 Range Queries . . . 7

2.6 Key-Value Stores . . . 7

2.6.1 Bucket-Key-Value Store . . . 8

2.6.2 Versioned Key-Value Store . . . 8

2.7 Existing NoSQL Databases . . . 8

3 Optimistic Replication Shim **11**

3.1 Causality Tracking . . . 12

3.2 Conflict Handling and Merging Semantics . . . 12

3.2.1 Delta Mutators . . . 14

3.3 Recovering Deleted Data . . . 15

3.4 Dissemination and Garbage Collection . . . 15

3.5 Extensions . . . 16

4 GeoD Implementation **17**

4.1 Overview . . . 17

4.2 GeoD Protocol . . . 18

4.3 API Translation . . . 19

4.4 Delta Management . . . 20

4.5 Propagator . . . 21

5 Emulating Versioning Support 23
5.1 Problem Definition . . . 23
5.1.1 Encoding Versions in Value . . . 23
5.1.2 Encoding Versions in the Key . . . 23
5.2 The VersionD Shim . . . 24
5.3 Improvements . . . 25
5.3.1 Garbage Collection . . . 25
5.3.2 Metadata Caching . . . 25
5.4 Previous Work . . . 26

6 Shared-Data Architecture 27
6.1 Architecture Overview . . . 27
6.2 Single Copy Per Chunk . . . 28
6.2.1 Deterministic Mapping . . . 28
6.2.2 Remote Update Signaling . . . 29
6.2.3 Traffic Duplication . . . 29

7 Conclusion 31

Bibliography 33

Chapter 1

Introduction

Running large Key-Value Stores out in the real world is a challenging task: processing thousands of requests per second on a petabyte-scale data set while keeping response times to a minimum in order to provide excellent user experience is no easy task. Geo-Replication is one of the key features of such systems – it allows the masking of failures while providing lower latency for operations, resulting in increased uptime and a better experience for users all around the globe.

However, not all key-value stores support this feature and, when available, it is usually not flexible enough to suit the user's needs – they usually employ classical forms of replication (synchronous or single-master), which are unsuited for large-scale applications.

A more scalable alternative to the traditional geo-replication mechanisms is optimistic replication (also known as lazy replication), a strategy in which replicas are allowed to commit operations locally and operate asynchronously. The main principle behind it is to do away with the total order of events, relaxing it into partial ordering. While it increases the complexity for programmers due to the introduction of concurrent events, it allows replicas to accept write operations at any time, including during network partition scenarios.

To this end we designed and developed GeoD, a shim that tackles the problem of providing optimistic replication to key-value stores. GeoD runs on top of independent key-value store instances and replicates all data asynchronously without any modification. It does so by intercepting requests from clients to the datastore, detecting the resulting internal state transitions and propagating them to the other sites. To save bandwidth, these transitions are buffered and aggregated before being sent. GeoD increases slightly the latency of requests, as it adds some processing delay to the critical path. Furthermore, the implementation described here is very general and covers only a small set of the common operations in the key-value stores.

The built-in design for conflict resolution follows two principles: it should be as familiar to the user as possible (i.e. it should behave similarly as if it was non-replicated), and should prevent unwanted loss of information – any deletion that is concurrent with a write to the same object will be ignored. Causality is tracked by resorting to version vectors. In case two values are written concurrently, the one with the largest physical timestamp wins (last-write-wins policy).

A typical desired feature of optimistically replicated systems is versioning – instead of storing a single value for each object, we may want to store multiple versions for logging or, more importantly, to support application-specific merge semantics. These allow for applications with complex merge semantics to have full control on how conflicting values are merged. To this end, we also developed VersionD, an upgrade to the basic GeoD that extends the client API with a versioning API.

VersionD enables legacy applications to make use of the built-in conflict resolution of GeoD while simultaneously allowing applications with specific merge semantics to enforce them. The downside to VersionD is the increased number of round-trips to the data store, which can impact negatively the latency of requests. The VersionD presented here also requires the data store to support prefix queries in order to provide some of its functionality (e.g. retrieving all the versions of a key), but simple workarounds can be designed.

Finally, we present an architecture for optimistically replicated data stores, by using two distinct storage components – a common data repository, shared by all the sites, and a private metadata repository, optimistically replicated at all sites. The main advantage of such an architecture is the massive reduction in synchronization overhead – instead of synchronizing a large file between all the sites, one just needs to broadcast it's metadata (typically a few bytes). This allows for synchronization messages to be very small, thus reducing bandwidth costs and propagation latency and, consequently the number of conflicting operations.

We discuss how to perform collision-free writes to the shared data repository without coordination. Efficient access to data is considered a sub-problem and is not covered in this work.

The contributions of this thesis are GeoD and VersionD, which allow to extend the feature set of key-value stores with optimistic replication. While GeoD only stores a single value per key, VersionD stores a set of values (called versions) to allow greater expressiveness in conflict resolution. We finish by presenting an architecture that allows for low-bandwidth low-latency propagation of updates between the different geo-locations.

Chapter 2

Background

This chapter introduces optimistic replication alongside with other relevant concepts.

2.1 Replication

Replication is a strategy in which multiple copies of an object are stored in multiple machines for increased availability, durability and either performance or reliability. By storing copies of data in separate machines, if one fails, a system can continue operating using the replicated data.

2.1.1 Objects, Replicas and Sites

An object is the fundamental entity in the system. They are the minimum granule of data in the system, and the minimal unit of replication. A replica is a copy of an object stored in a site, or in a computer. A site may store replicas of multiple objects, but I will use the terms replica and site interchangeably, as the algorithms discussed here manage each object independently.

2.1.2 Geo-Replication

Geo-replication is a specific use-case of replication in which the replicas of an object are scattered over different geographic sites. In this scenario we assume network characteristics poorer than in the single-site scenario: the latency between the replicas is a few orders of magnitude higher, the bandwidth is much more limited and link failures are more unpredictable.

By having a copy close to clients, they will experience improved system uptime and lower access latency [78]. One of the most popular use-cases is in the Domain Name System [26], where having multiple copies of the records of a name results in near-perfect uptime, even in the presence of denial-of-service attacks.

2.1.3 Optimistic Replication

Optimistic replication [71, 72] is a group of techniques for efficient data-sharing in wide-area or mobile environments. They differ from traditional *pessimistic* database replication techniques [42] in the way they handle concurrency. While pessimistic algorithms synchronously coordinate replicas during accesses and block other users during an update, optimistic replication allows data to be accessed without prior synchronization. Updates are propagated in the background, and conflicts are handled after they are detected. The advantages include improved availability, as applications continue to make progress even under the presence of arbitrary network failures, and scaling linearly to a large number of replicas due to the reduced synchronization required between the sites.

2.2 Time and Ordering of Events

In a local environment, time and ordering is trivial. For time, we can rely on the local physical clocks, and use them to provide ordering of events in the system. However, in a replicated environment, clocks are unreliable [62, 75]. Logical clocks [38, 59] were developed to tackle the problem, but only provide a partial ordering of events.

Vector clocks [23, 51] provide a mechanism to figure out a partial ordering of events based on their causality. Version vectors [14, 15, 17, 46, 58, 61, 68] are similar to vector clocks; however, their purpose is not to order the events in a distributed system, but to track the causality between the events in replicas. These have been shown to be optimal [31] in size – to unequivocally track causality there must be one entry per source of concurrency. More recently, interval tree clocks [16] were introduced, which generalize the above mechanisms for dynamic systems, where processes can be created or retired in a decentralized fashion.

2.3 Consistency and Conflicts

The definition of conflict is inseparable from that of consistency. A conflict occurs when a set of concurrent operations violate consistency. For example, if two replicas concurrently execute two distinct *put* operations to the same key with different values, the one executed the latest will replace the value of the earliest. If the operations are not applied in the same order, the replicas will diverge indefinitely. The system is said to be consistent if all the replicas are in the same logical state.

In a single-replica environment, its state management and storage is centralized. Independently of the number of operations, concurrent or not, there is at all times a consistent and unique state – each user will see a consistent view of the data. In single-master replicated environment (e.g. DNS [26]), replicas are kept consistent (except for brief periods due to message propagation delay), but remain consistent in the long run.

However, on multi-master systems, replicas may diverge without bound if operations conflict. Let us take a replicated key-value data store as a concrete example. If two replicas concurrently set the value of an object to different values, what should be the final outcome? If two replicas execute the operations in the order they receive them (i.e., first the local update, and later the one from the remote replica), they will diverge into different states.

There are two ways to overcome this: either synchronize the replicas (by ordering the conflicting updates), or designing the operations in such way that conflicts do not occurr. We call these two families of models *strong consistency* and *weak consistency*, respectively.

Under a network partition scenario, the system can be either completely available, or kept strongly consistent. This has been popularized as the CAP theorem[27, 39, 41, 66, 80, 81].

2.3.1 Strong Consistency

Strong consistency requires some form of global agreement on the order of updates, and the user will have a similar experience to that of a single-master replicated system. All replicas return the same value when queried (except for the message propagation delay mentioned before). However, operations must perform synchronization in order to decide on the outcome of concurrent conflicting operations.

2.3.2 Weak Consistency

Weak consistency allows for replicas to execute updates without the constant synchronization imposed by strong consistency – we call it *disconnected operation*. Weak consistency, however, imposes some limitations in the outcome of concurrent sets of operations. In order to provide disconnected operation, all the operations must be designed in a way so that they are associative, commutative and idempotent [74].

In addition to this limitation, some of the system's invariants may not be respected if we allow all operations to be executed without synchronization [24, 52]. One such example of an invariant is *'the balance of an account must be non-negative'* in a bank application: different replicas can withdraw amounts whose sum is greater than the total balance, causing it to be negative after synchronization.

We briefly describe relevant weak consistency models.

Eventual Consistency

Eventual consistency simply states informally that *if no updates are made to an object, all replicas will eventually converge.* It does not define the procedure that achieves convergence, nor any methodology. It is the weakest consistency model. The other models here discussed can be seen as strengthenings of eventual consistency.

Strong Eventual Consistency and CRDTs

Strong eventual consistency states that *replicas that have received the same set of updates are in the same state.*

A Conflict-free Replicated Data Type (CRDT [74]) is a principled approach into designing a replicated data type providing strong eventual consistency semantics. CRDTs can be designed using two frameworks – *operation-based* and *state-based.* In the operation-based framework, concurrent operations are made commutative. In the state-based framework, states are merged in such a way they form a monotonic semi-lattice. The two approaches have been shown to be equivalent, as one can emulate the other. Delta-CRDTs [18] are an approach for deriving efficient state-based CRDTs, by computing how the state changes for each of the operations executed by the user.

Causal Consistency

Causal consistency employs causal (happens-before [13]) order [73] between the updates, for example through the use of Version Vectors [46]. Updates that are ordered do not conflict – all replicas will apply them in the same order (similar to what is accomplished by strong forms of consistency). However, two concurrent operations may still cause a conflict (as they are not causally ordered). In this case, the resolution of the conflict is not defined.

Causal+ Consistency

Causal+ consistency is a hybrid of Causal Consistency and Strong Eventual Consistency – uses version vectors to capture the causal order of updates, but in case of a conflict, employs automatic conflict resolution. One such example of conflict resolution is in GentleRain [36], where in addition to having one version vector per operation, a physical timestamp is also attached as a tie-breaker to decide on a total ordering of operations.

2.3.3 Hybrid Consistency Models

In order to satisfy global invariants, hybrid consistency models have been developed. Red-Blue Consistency [24, 52] splits the set of operations into *red operations* (those that may violate global invariants if executed concurrently and therefore must be executed with coordination between all replicas) and *blue operations* (that can be executed locally before being later asynchronously propagated). Naturally, red operations are unavailable under network partitioning scenarios, the system being able to execute blue operations only.

2.4 Distributed Hash Table

A Distributed Hash Table (DHT) is a peer-to-peer (P2P) systems that are scalable, robust and self-organizing. They provide a lookup service similar to that of a hash table. Nodes are arranged in a ring, each taking responsibility for $1/n$ of the keyspace on average. Nodes are assigned a *position* in the ring (e.g. by hashing their identifier) and take responsibility for the interval between its *predecessor*'s position (the node with the position before in the ring) and its own. Responsibility of mapping the keys to values is distributed among the nodes in such a way that if the set of participants changes, the system recovers with minimal disruption.

Chord [77] is among of the first implementations of a DHT. Chord employs consistent hashing [47] to partition its 160-bit key space and to balance the load between the participating nodes.

2.5 Range Queries

A range query is a common database operation that retrieves all records where some key is between an upper and a lower boundary. In order to these queries efficiently, we require an efficient indexing of the keys.

Some storage systems opt to store all the records sorted. For example, in BigTable [29] the key space is partitioned into tablets, each storing a consecutive subset of the possible keys, and each tablet is mapped onto a single machine. This technique may cause hot-spots, as the more popular regions of data will fall on the same set of machines.

Other systems have a separate search structure called an *index*. Indices are a copy of the list of keys that can be searched efficiently. A classic data-structure used in the implementation of indices is the B-tree [25]. These, however, are unsuited for concurrent operation – for correctness, no more than a single access at a time can be done to a given subtree. No variant of these structures is known to work well in a multi-master replication environment.

DHTs are excellent candidates to be at the core of a key-value store. However, due to the lack of locality, they require a separate index to be built in order to support efficient range queries [19]. For this purpose, specialized data structures like the Prefix Hash Tree [69] were developed. However, all of them have the same limitations of the classical counterparts – concurrent updates to the same subtree may violate the structure's invariants and require updates to be serialized.

2.6 Key-Value Stores

A Key-value store is the simplest type of NoSQL database [63]. It is a simple map: store a set of objects, each uniquely identified by a key. We will use *oid* (**O**bject **Id**entifier) to refer to the key of an object. The core API provided by a Key-Value store can be seen in table 2.1.

Operation	Description
addKey(oid, value)	Sets the contents of the object *oid*
deleteKey(oid)	Deletes the object *oid*
getKey(oid)	Retrieves the contents of object *oid*

Table 2.1. Core operations on Key-Value data stores

2.6.1 Bucket-Key-Value Store

The Bucket-Key-Value Store (or Bucket Store) model augments the Key-Value Store by introducing the concept of *buckets*. A bucket is a logical container for storing objects that organizes the namespace at the highest level, each with a unique identifier. In practical terms, each can be seen as an independent instance of a key-value store. This is the model followed by Amazon S3 [1]. The core API provided by Bucket-Key-value stores can be seen in table 2.2. Throughout the report we will use *bid* (**B**ucket **Id**entifier) to refer to the identifier of a bucket.

2.6.2 Versioned Key-Value Store

A Versioned Key-Value Store not only stores and exposes the latest version of the value associated with a given key, but keeps a history of values that have been assigned to it, called *versions*. Each version encodes the state of the object at a point in time and space. Versioning support is explicitly part of the data model and exposed by the API. The versioning API can be seen in table 2.3.

2.7 Existing NoSQL Databases

Key-Value stores are the simplest of NoSQL databases. We briefly discuss relevant details of some of the current implementations.

Operation	Description
createBucket(bid)	Creates a new bucket with the identifier *bid*
dropBucket(bid)	Drops the bucket with the identifier *bid*
addKey(bid, oid, value)	Sets the contents of the object *oid* in the bucket *bid*
deleteKey(bid, oid)	Deletes the object *oid* in the bucket *bid*
getKey(bid, oid)	Retrieves the contents of the object *oid* in the bucket *bid*

Table 2.2. Core operations on Bucket-Key-Value data stores

Operation	Description
addVersion(bid, oid, vid, value)	Adds a version *vid* of object *oid* in bucket *bid*
deleteVersion(bid, oid, vid)	Deletes version *vid* of object *oid* in bucket *bid*
getVersion(bid, oid, vid)	Retrieves version *vid* of object *oid* in bucket *bid*
getLatest(bid, oid)	Retrieves the set of latest versions of object *oid* in the bucket *bid*

Table 2.3. Versioning API for Bucket-Key-Value Stores

Amazon Dynamo

Dynamo [33] is Amazon's highly-available key-value store. Amazon uses a highly decentralized, loosely coupled service-oriented architecture consisting of hundreds of services. For the shopping cart service, they require storage technologies that are always available: "customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornadoes." Therefore, this system requires a data-store that is always both readable and writable, and that it's data needs to be available across multiple data centers.

Dynamo, however, targets applications that require only key/value access with primary focus on high availability. No powerful features such as prefix queries are required by the applications, nor are they provided by Dynamo.

Many traditional data stores execute conflict resolution during writes and keep the read complexity simple. However, in such systems, writes may be rejected if the data store cannot reach all (or a quorum) of the replicas at a given time. Dynamo targets the design space of an "always writable" data store. This requirement forces the pushing of complexity of conflict resolution from the writes to the reads in order to ensure that writes are never rejected. In Dynamo, conflict resolution can be done either by the data store or by the application. If done by the data store, only simple policies can be used (such as last-writer-wins). However, if delegated to the application, more complex merge policies can be used with the added complexity of managing this merging.

Voldemort

Voldemort [9] is a distributed key-value storage system, used at LinkedIn [5] by numerous critical services powering a large portion of the site, based on Amazon Dynamo.

COPS

COPS [54] is a key-value store providing Causal+ Consistency. COPS can enforce causal dependencies between keys stored across an entire cluster, rather than a single server as previous systems. COPS checks whether causal dependencies between keys are satisfied in the local cluster before exposing writes. Further, in COPS-GT (an extension of COPS) they introduce read-only transactions in order to obtain consistent views of multiple keys without locking.

Netflix Dynomite

Dynomite [8] is an open-source project by Netflix that provides replication for key-value stores. It is a generic Dynamo implementation that can be used with many different key-value pair storage engines. It supports multi-site replication and is designed for high-availability. It uses an operation-based approach: changes broadcasted to other replicas take the form of a log of operations. However, it provides no consistency guarantees whatsoever – operations may be reordered, or executed more than once, and no efforts are made to change their semantics to make them associative, commutative nor idempotent. Only applications that do not require any consistency guarantees may be used with Dynomite, such as publishing time series data.

Riak

Riak [10] is a distributed key-value store offering high scalability, fault tolerance and scalability. Its design is very influenced by Amazon Dynamo's. In addition to storing binary objects, Riak allows users to choose among several types of CRDT objects, whose convergence is then automatically handled by Riak.

Amazon S3

Amazon Simple Storage Service (S3) [1] is a bucket-key-value cloud storage service. Amazon S3 supports retrieval of an object by using the full object key or using a prefix.

GentleRain

GentleRain [36] is a causally consistent replicated data-store, providing throughput comparable to that of an eventually-consistent data-store. GentleRain

SwiftCloud

SwiftCloud [82] is a data storage system for cloud platforms that spans both client nodes (that cache a subset of the objects) and datacenter servers (that replicate every object).

Chapter 3

Optimistic Replication Shim

In this chapter we present and detail the design of GeoD, an optimistic geo-replication shim for versioned bucket-key-value stores. It provides the following:

Availability Even in the event of node or network failures, the system must remain available. Any distributed system in the presence of network failures faces a trade-off between availability and consistency [27, 39, 41, 66, 80, 81]. With this in mind, for writes to succeed under an arbitrary inter-DC network partition scenario, the different replicas must be allowed to diverge, albeit temporarily. For this reason, we employ a weak consistency model.

Perseverance of read-only functionality The set of read-only operations (i.e. those not causing any changes in the internal state) that can be provided by the key-value must remain available after applying the shim. As an example, if a key-value store can support prefix queries, it must also be able to provide them when geo-replicated.

No mechanism interference Some systems fulfill the previous requirements, but only at the expense of some other features. For example, Cassandra fulfills the availability requirement but, in order to provide range queries, it requires swapping the consistent hashing [47] function for an order-preserving hash function in the mapping of objects into storage nodes, which may cause hotspots during data placement.

Separation of Concerns The underlying data store requires no modification and replication should be provided in the form of a shim, uniquely by intercepting and injecting messages in the system. The solution should be general enough to be used by any key-value stores providing the same storage model (or a superset) and the same API (or a subset).

GeoD intercepts requests from clients, and determines what transitions happening at the data store level, propagating them to the other replicas. GeoD's semantics guarantees that no conflicts will arise from concurrent operations by employing automatic conflict resolution – the outcome of the operations is designed to be associative, commutative and idempotent. The separation of concerns in the GeoD architecture is represented in figure 3.1.

In addition to convergent conflict handling, GeoD tracks the causality between operations. If an operation *op_1* happens before another operation *op_2* and they both target the same object or bucket, the outcome of operation *op_1* is discarded.

The following section will detail the design of GeoD and discusses possible variations.

3.1 Causality Tracking

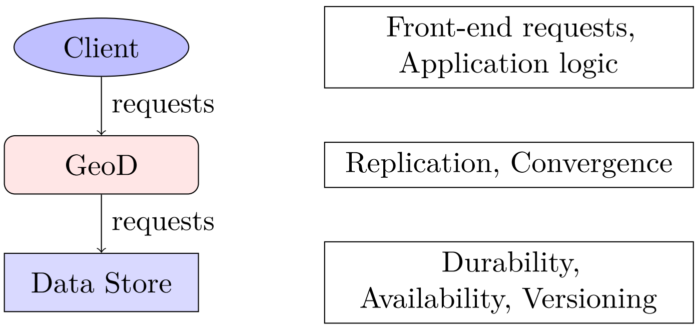
GeoD employs the use of vector clocks to capture the causality relationship between the different versions of an object. We define the causality relationship in the system as the happened-before order: if one replica sees a given (local or remote) update *u1*, by definition all other updates issued locally happen after *u1*. If two updates are not causally related, they are concurrent (as previously described). More formally, the set of updates issued to the system globally is a partially ordered set.

3.2 Conflict Handling and Merging Semantics

Concurrent operations may conflict with one another. As an example, if a client changes the value of an object while another deletes it, the semantic outcome is not well-defined. In the presence of concurrent updates to the same object, we want to be able to enforce strong eventual consistency semantics. In order to achieve this, we must make all the operations associative, commutative and idempotent, while maintaining their meaning.

The choices available between *create* and *drop*, as well as between *add* and *delete* are simple: one can either opt for a *delete-wins* approach (simpler to implement; which favors the deletion of information) or an *add-wins* approach (preferred; avoids

Figure 3.1. Separation of concerns in the GeoD architecture, where a shim layer replicates the underlying data store by intercepting requests from clients


Client
requests
GeoD
requests
Data Store
Front-end requests, Application logic
Replication, Convergence
Durability, Availability, Versioning

deleting information if it is concurrently accessed). We chose the latter, as it proves more familiar to reason with, and avoids accidental removal of information.

There is yet another conflicting case – when two users concurrently set the value of a key, what should be its result? Clearly, the ideal value should be a merge of both values, which is dependent on the context on which it is inserted (i.e. what the abstract binary value represents in the context of the application storing it). There are a number of options we can choose from:

Last Writer Wins Although we cannot rely on physical clocks to ensure consistency, they can be used as a tiebreak for conflicts. Consider the following composition: [physical clock][replica ID]. This will allow us to provide linearizability between all updates, from all replicas, at each of the replicas. This means that we will always be able to decide that one update is more recent than the other, effectively discarding the oldest. This is the approach taken by most eventually-consistent storage systems, such as Amazon S3.

CRDT-Merge Function Any associative, commutative and idempotent function is enough to merge conflicting updates. Common examples are maximum, minimum, summation and multiplication functions. This provides potentially better results than the last-writer-wins, but is very context-dependent. Although we can allow the user to decide how he wants to merge the keys of a bucket, the data-store is also limited to using very simple policies, which may not be the optimal for each application.

Operation Pairs	**createBucket(bid)**	**dropBucket(bid)**
createBucket(bid)	Bucket *bid* is created	Bucket *bit* is created; Drop operation ignored
dropBucket(bid)	Bucket *bit* is created; Drop operation ignored	Bucket *bid* is dropped
setKey(bid, oid, value) **deleteKey(bid, oid)**	Bucket *bid* is created	Bucket *bit* is created; Drop operation ignored

Table 3.1. Merge semantics for bucket operations

Operation Pairs	**setKey(bid, oid, value)**	**deleteKey(bid, oid)**
setKey(bid, oid, value)	?	Object *oid*'s value is set
deleteKey(bid, oid)	Object *oid*'s value is set	Object *oid* is deleted

Table 3.2. Merge semantics for object operations

Delegate to Application As this provides the best results, at the cost of increased complexity for the programmer. However, applications may not want or be able to make use of this feature – if such capabilities are provided, they should be optional.

Some clients may not want to deal with conflict resolution, specially legacy applications. However, other clients may want to deal with concurrent conflicting operations.

The ideal solution is to provide both: have a merge policy based on last-writer-wins attached to the normal *get* operation, but augment the API with a *getConcurrent* operation, which returns multiple values (the set of most recent concurrent versions of the object). To achieve this, a vector clock is attached to each operation as the primary tiebreak. In addition, every operation is tagged with a physical timestamp, generated by the clock at the local replica. Therefore, physical clocks of all of the replicas should be kept in sync.

In addition to storing multiple versions of a given object (one corresponding to each value written), an index of versions and their order must be kept and maintained. This responsibility is offloaded to the data store.

3.2.1 Delta Mutators

There are two approaches to propagate the changes done to a replica: a state-based, and an operation-based. The state-based approach involves sending the entire state of the replica – this can quickly become a huge overhead, as the state grows too big. On the other hand, one may send the log of local operations, which is not optimal – overwritten states can be discarded safely.

To avoid these limitations, we use the δ-CRDT approach: a state-based approach that encodes only the most recent changes (i.e. the ones that the destination replica is not aware).

In this section we describe the *δ-mutators* [18, Definition 1], one for each update operation in the system. The computation of mutators is tightly related to the data model (bucket-key-value store) and operations provided to manipulate it (bucket creation and deletion, object creation and deletion).

To follow the δ-CRDT specification, each of the operations must be associated with a mutator function, which encodes the internal state changes caused by each instance of the operation. For example, if a user creates bucket B, the mutator must encode that bucket B was created, and that it does not create conflicts with any other concurrent operation in any site.

Table 3.3 and table 3.4 show how each instance of an operation affects the internal state of the data store, along with the preconditions for it to complete successfully.

Operation	Pre-conditions	Effect
createBucket(bid)	Bucket *bid* does not exist	Bucket *bid* is created
dropBucket(bid)	Bucket *bid* exists	Bucket *bid* is dropped

Table 3.3. Mutators for operations on buckets

Operation	Pre-conditions	Effect
setKey(bid, oid, value)	Bucket *bid* exists	Value of object *oid* in bucket *bid* is set to *value*
deleteKey(bid, oid)	Bucket *bid* exists	Object *oid* is removed from bucket *bid*

Table 3.4. Mutators for operations on objects

3.3 Recovering Deleted Data

A problem arising with the previously discussed merge semantics is the need to undelete buckets and their contents in case a concurrent *createBucket* or object update operation is issued. Therefore, the actual deletion of data must be delayed until we are sure no concurrent updates happened and all replicas acknowledge the delete. Whenever a bucket is deleted, we only mark the bucket and its contents as deleted. All local subsequent requests to the marked-as-deleted bucket result in an error, until recreated.

The actual deletion at a given replica occurs when it receives acknowledgement from all other replicas that they received the deletion, and no concurrent creation or update of the bucket happened. The fact that deleted buckets may reappear after some time won't be accounted in most user's mental models, and can be effectively considered *strange behaviour*. An alternative is to synchronize bucket operations, eliminating their concurrency.

3.4 Dissemination and Garbage Collection

All replicas periodically broadcast their current-epoch updates to all other sites, along with a version vector encoding what sets of updates it has seen. As soon as all replicas asynchronously acknowledge they have seen a given set of updates, those updates can safely be garbage-collected.

As long as there is some connectivity and sufficient bandwidth (i.e. the average bandwidth between replicas exceeds the average amount of information to be sent), replicas are guaranteed to make progress, and information can be garbage-collected before a too-large amount is accumulated.

3.5 Extensions

In this section we discuss possible enhancements that can be added to GeoD, based on existing literature.

GeoD encompasses only the core operations of key-value stores. However, it can be easily extended to richer data models, such as column-family stores (as Eiger [55] does), or even incorporating the mechanics of Indigo [24] to support arbitrary applications.

Incorporating Indigo would also allow GeoD to support global invariants. For example, in a given context the key-value pairs in a bucket may represent account numbers and their balance. If we want to enforce that all accounts must have positive balance, withdrawals cannot be done without synchronization.

In addition to seeing objects as black-boxes, one could extend GeoD to be a Key-CRDT store [32], as in SwiftCloud [82]. This would allow for rich conflict-resolution policies at the data-store level, freeing developers from the burden.

Chapter 4

GeoD Implementation

4.1 Overview

In this section we detail the components of GeoD with enough detail to produce a real-world implementation of the system. The GeoD implementation has three major components: an interference component dependent on the data-store and its API, a delta-management module and a propagator. The internal architecture of GeoD can be seen in figure 4.1.

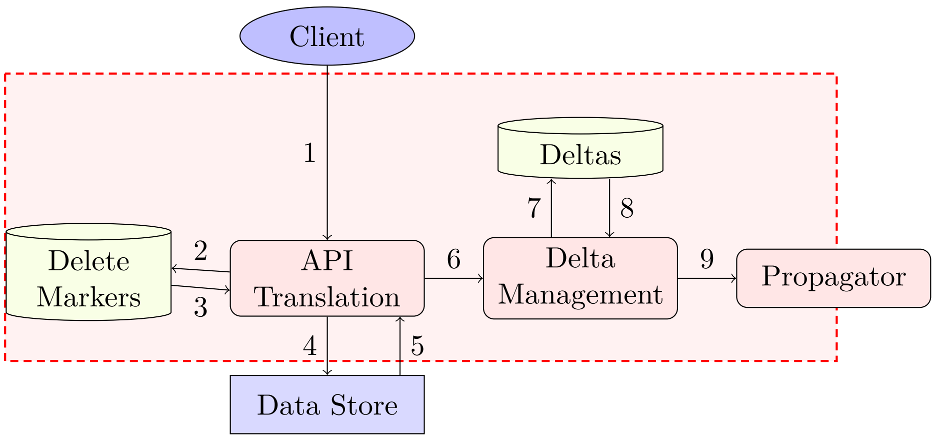
Client requests are intercepted and analyzed at the *API Translation* component – mainly to check if the user is accessing items that have been marked as deleted. Any delete operations issued by the user are removed from the request, and are instead written into the *Delete Markers* database. After these two tasks complete, the request is then forwarded to the data store.

Once the data store returns a response, it is intercepted and modified: the result of any *delete* operations that were not sent to the data store must be added to the response body before forwarding it to the client. At this stage, one must infer the internal changes that occurred in the data-store, using our δ-CRDT abstraction – each of the operations is translated into its corresponding set of state changes and sent to the *Delta Management* module.

The *Delta Management* is responsible for taking these mutators and merging them into a global *delta* object, which encodes all the internal state changes that happened in the current epoch. Periodically, the *Delta Management* will push the *delta* object and forward it for propagation to the other sites, and start a new epoch (i.e. reset the *delta* object to a *no-changes* representation, incrementing the local version vector).

Upon reception of remote updates, they are compared with the latest local versions. If the remote version is older, or is a concurrent *delete* or *drop*, it is discarded; otherwise, they are merged in the local data store. Any *delete* or *drop* operations that have not been acknowledged by all replicas are instead converted to a delete marker, rather than applied at the data store itself. If a non-delete operation is concurrent with a delete marker, it removes the delete marker.

Figure 4.1. Internal components and relationships of GeoD. The arrows illustrate the path followed by each request processed.



4.2 GeoD Protocol

In this section we formalize the protocol used by clients to communicate with GeoD. GeoD exposes an API equivalent to that of a Bucket-Key-Value store. Each GeoD request can be seen as a log of operations. The grammar defining GeoD requests to data stores can be seen in definition 4.2.1. Different key-value stores have different but equivalent APIs – they can be translated into GeoD's API. Some may allow requests to a single bucket, while others may encode the bucket ID in the object requests, etc.

Definition 4.2.1. BNF specification of the GeoD protocol exposed to clients

$$\begin{array}{lcl}
\langle \textit{request} \rangle & ::= & \langle \textit{commands} \rangle \\
 & & \\
\langle \textit{commands} \rangle & ::= & \langle \textit{commands} \rangle \ \langle \textit{command} \rangle \\
 & | & \langle \textit{command} \rangle \\
 & & \\
\langle \textit{command} \rangle & ::= & \text{create bid} \\
 & | & \text{load bid} \\
 & | & \text{drop bid} \\
 & | & \text{add bid oid value} \\
 & | & \text{get bid oid} \\
 & | & \text{delete bid oid}
\end{array}$$

In the examples here presented we will use JSON in place of a syntax tree for clarity. An example of a request to GeoD can be seen in example 4.2.1.

Example 4.2.1. A GeoD request in JSON format. The request adds the key-value pairs (*key1*, *newval1*) and (*key2*, *newval2*) to an already-existing bucket *bucket1*.

```
[
    {
        "op": "add",
        "bid": "bucket1",
        "oid": "key1",
        "value": "newval1"
    },
    {
        "op": "add",
        "bid": "bucket1",
        "oid": "key2",
        "value": "newval2"
    }
]
```

4.3 API Translation

The *API Translation* component is responsible for intercepting the requests to the data-store. For every command that changes the data-store's internal state, it emits a *mutator* (section 3.2.1) encoding the state transition, which is then merged into the deltas to be broadcasted to the other sites. The mutator's structure is defined in definition 4.3.1. It must also be able to do the inverse – to generate requests to the data-store based on mutators. This is vital to the merging of remote updates.

Definition 4.3.1. BNF specification of the list of δ-mutators

$$
\begin{array}{lcl}
\langle \textit{mutators} \rangle & ::= & \langle \textit{buckets} \rangle \\[1ex]
\langle \textit{buckets} \rangle & ::= & \langle \textit{bucket} \rangle\ \langle \textit{bucket} \rangle \\
 & | & \langle \textit{bucket} \rangle \\[1ex]
\langle \textit{bucket} \rangle & ::= & \text{bid}\ \langle \textit{exists} \rangle\ \langle \textit{objects} \rangle \\[1ex]
\langle \textit{objects} \rangle & ::= & \langle \textit{objects} \rangle\ \langle \textit{object} \rangle \\
 & | & \langle \textit{object} \rangle \\[1ex]
\langle \textit{object} \rangle & ::= & \text{oid}\ \langle \textit{exists} \rangle\ \text{value} \\[1ex]
\langle \textit{exists} \rangle & ::= & \text{true} \mid \text{false}
\end{array}
$$

In addition to the translation routines, it also requires to keep track of deleted items – if a user requests access to one of these, a *not found* error response must be generated and sent back to the user. Conversely, if a user deletes one of these items, a delete marker must be generated for future reference. In our implementation we only need to keep track of deleted buckets. The data structure's organization can be seen in definition 4.3.2.

Algorithm 4.1 Handling Requests from Clients

```
function HandleClientRequest(request)
    deleteMarkers ← GetDeleteMarkers()
    dropOperations ← $\emptyset$
    for all bid ∈ request do                         ▷ Check accesses to deleted buckets
        if bid ∈ deleteMarkers then
            return ErrorMessage()
    for all operation ∈ request do                   ▷ Remove drop operations
        if operation.op = *drop* then
            dropOperations ← operation
    response ← SendToDatastore(request)
    response ← includeDeleteMarkers(dropOperations, response)
    mutators ← GenerateMutators(request, response)
    if response.status = *OK* then
        deleteMarkers ← deleteMarkers ∪ dropOperations
        MergeLocalDeltas(mutators)
    return response
```

Definition 4.3.2. BNF specification of the delete markers structure

```
⟨delete-markers⟩   ::= ⟨delete-markers⟩ ⟨delete-marker⟩
                     |  ⟨delete-marker⟩

⟨delete-marker⟩    ::= bid ⟨vv⟩

⟨vv⟩               ::= ⟨vv⟩ ⟨vventry⟩
                     |  ⟨vventry⟩

⟨vventry⟩          ::= ⟨site-id⟩ ⟨vvindex⟩
```

4.4 Delta Management

At all stages we need to keep a complete summary of all the incremental changes done in the data store. We call this summary a *delta*. The Delta Management component is responsible for merging the mutators into a global *delta* object encoding the union of the incremental state changes. The deltas are a fast-growing set of data with no size bounds that needs to be persisted.

Definition 4.4.1. BNF specification of the deltas

```
⟨deltas⟩           ::= ⟨vvindex⟩ ⟨buckets⟩

⟨buckets⟩          ::= ⟨buckets⟩ ⟨bucket⟩
                     |  ⟨bucket⟩
```

$\langle \mathit{bucket} \rangle$	::=	bid $\langle \mathit{vv} \rangle$ $\langle \mathit{exists} \rangle$ $\langle \mathit{objects} \rangle$
$\langle \mathit{objects} \rangle$	::=	$\langle \mathit{objects} \rangle$ $\langle \mathit{object} \rangle$
	|	$\langle \mathit{object} \rangle$
$\langle \mathit{object} \rangle$	::=	oid $\langle \mathit{versions} \rangle$
$\langle \mathit{versions} \rangle$	::=	$\langle \mathit{versions} \rangle$ $\langle \mathit{version} \rangle$
	|	$\langle \mathit{version} \rangle$
$\langle \mathit{version} \rangle$	::=	$\langle \mathit{vv} \rangle$ $\langle \mathit{exists} \rangle$ value
$\langle \mathit{exists} \rangle$	::=	boolean
$\langle \mathit{vv} \rangle$	::=	$\langle \mathit{vv} \rangle$ $\langle \mathit{vventry} \rangle$
	|	$\langle \mathit{vventry} \rangle$
$\langle \mathit{vventry} \rangle$	::=	site-id scalar-clock

4.5 Propagator

The propagator is the component in charge of sending and receiving messages from all other participating GeoD processes in the replication group. The actual implementation details of this component are not relevant for the application presented here and are therefore omitted.

The propagator should implement a broadcast protocol [30, 34, 60] to periodically propagate its local deltas to the other sites. As GeoD's only requirement for the inter-site network is that message integrity is preserved, any broadcast algorithm suffices – no need for reliability nor atomicity.

In addition to the broadcast protocol, it needs to keep a consistent view of the members of the system.

This chapter detailed the necessary core components to develop an optimistic replication shim. In the following chapter we discuss a possible deployment scenario for GeoD with heterogeneous storage nodes, where some nodes store data and others only store metadata.

Chapter 5

Emulating Versioning Support

Big Data applications favor the utilization of key-value stores over traditional relational database systems for their improved ability to scale. These usually map each object to a single value, not supporting multiple versions for a given key. However, applications that require complex merge semantics might want to be exposed to object versioning. One of such applications is GeoD. This chapter presents how to emulate versioning support by adding a shim on top of key-value stores that only support a single value per object.

5.1 Problem Definition

We formulate the problem of enabling multiple versions per object on top of an arbitrary strongly-consistent bucket-key-value store providing the core API (tables 2.1 and 2.2) and extending it with a versioning API (table 2.3).

As we are trying to map three parameters (key, value and version) into an API that accepts two (key and value), there are two possible approaches:

5.1.1 Encoding Versions in Value

The simplest solution is to encode all the versions in the value. This changes the mapping of each key to an object, to having each key mapped to a list of versions of that object. The list of versions is a black-box for the key-value store, and is externally managed. However, for objects with many versions this may become impractical due to the large amount of data retrieved in every operation. In practice, all the read operations must retrieve all the versions of the object, and all write operations must read the current list of versions and append a new version to it. This may add significant network utilization, as well as increased latency.

5.1.2 Encoding Versions in the Key

Another solution is to encode the version in the key, by partitioning the actual keyspace into disjoint logical keyspaces. This addresses the previous network utilization problems, as one is able to retrieve each version individually, and may append new versions without reading the previous ones. Some data stores, however, may impose a limit on the length of a key.

For example, if the maximum key length is 128 bytes and the user specifies a key with length 120 bytes, only 8 bytes are left to encode the version. If the version cannot be encoded in the remaining 8 bytes, storing or accessing such version results in an error.

In practice, this can be lifted by simply shortening the length of keys that can be specified by the user, thus reserving a fixed (small) amount of bytes to specify the version. Although this imposes a limit on the number of versions, using techniques such as base64 [45] can encode a large amount of versions in little space.

5.2 The VersionD Shim

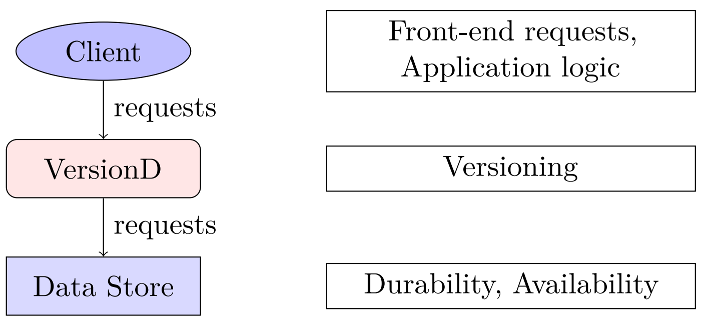
In this section we introduce VersionD, a versioning shim for key-value stores providing the core API. VersionD encodes the version identifiers in the key part of the object. For design simplicity, we consider assumption 1 in the design of VersionD. The separation of concerns in the VersionD architecture can be seen in figure 5.1.

VersionD handles versioning in a way similar to that of Amazon S3 [1]. The *add* operation replaces the current value by appending a new version to the object. The $add_{version}$ does the same, but the version assigned to the value is arbitrary – it may be more recent, older, or concurrent. The *delete* operation appends a *delete marker* to the object – the object is marked and shown as removed, but no data is lost. The $delete_{version}$ removes a specific version from the data store.

Due to versioning and causality, three *get* operations are provided. The normal *get* returns the result of merging all the latest values of the object. The $get_{version}$ simply returns a specific version of an object. get_{latest} returns all the latest values of the object – the set of values corresponding to all the latest concurrent versions.

Assumption 1. The underlying key-value store imposes no limit on the length of keys.

Figure 5.1. Separation of concerns in the VersionD architecture, where a shim layer augments the API of the underlying data store by incorporating versions in the mapping of objects



It is clear that we need to keep some metadata about the versions of an object, in addition to their actual values. In our implementation we only need to keep track of the set of latest versions of an object (i.e. those that are not older than any other version), but other operations may be supported if needed. For example, one may want to keep the full list of versions of an object to support operations such as *listAllVersions(oid)*, providing clients with the listing of versions being stored.

Table 5.1 describes how client operations map into the underlying core operations. In order to partition the bucket space to store different kinds of information without them conflicting, we add a prefix (algorithm 5.1). This prefix behaves as a selector for the partition we want to access.

Algorithm 5.1 Key Mapping Functions. The symbol $\|$ denotes the concatenation of strings. The symbol $\bullet$ is a delimiter used in the key that does not occur in *oid*, or is escaped. In a practical implementation the actual prefixes used can be optimized to fit in fewer bits.

```
function $M_{latest}(oid)$                ▷ Maps a key to its set of concurrent versions
    return '$LATEST$' $\|$ $\bullet$ $\|$ $oid$
function $M_{versions}(oid)$                                ▷ Maps a key to its versions
    return '$VERSIONS$' $\|$ $\bullet$ $\|$ $oid$
function $M_{version}(oid, version)$          ▷ Maps a (key, version) pair to its value
    return $M_{versions}(oid)$ $\|$ $\bullet$ $\|$ version
```

The mechanism presented here is sufficient to provide the desired versioning features, and combined with GeoD makes it possible to transform any key-value store into a fully-featured weakly-consistent key-value store, with customizable merge semantics and versioning support.

5.3 Improvements

We present some improvements for VersionD that would be useful in practice, but are not relevant to the core functionality of the system.

5.3.1 Garbage Collection

The implementation presented here does not perform garbage collection of old versions. Most applications don't make use of obsolete versions and, in these cases, they can be deleted. Garbage collection can be done easily, by slightly modifying the mapping between client operations and data store operations. No further storage is required.

5.3.2 Metadata Caching

Most of the data store routines start by retrieving the required metadata associated with the key in the request. In our case, the metadata is a potentially small list of

Client Operations	**Data Store Operations**

add(*oid*, *value*)	*version* ⇐ *NewVersion()*
	add($\mathrm{M}_{\text{latest}}$(*oid*), {*version*})
	add($\mathrm{M}_{\text{versions}}$(*oid*, *version*), *value*)

$\text{add}_{\text{version}}$(*oid*, *value*, *version*)	*latest* ⇐ latest(get($\mathrm{M}_{\text{latest}}$(*oid*)), *version*)
	add($\mathrm{M}_{\text{latest}}$(*oid*), *latest*)
	add($\mathrm{M}_{\text{versions}}$(*oid*, *version*), *value*)

delete(*oid*)	*version* ⇐ *NewVersion()*
	add($\mathrm{M}_{\text{latest}}$(*oid*), {*version*})
	add($\mathrm{M}_{\text{versions}}$(*oid*, *version*), *deleteMarker*)

$\text{delete}_{\text{version}}$(*oid*, *version*)	add($\mathrm{M}_{\text{latest}}$(*oid*), {*version*})
	add($\mathrm{M}_{\text{versions}}$(*oid*, *version*), *deleteMarker*)

get(*oid*)	get($\mathrm{M}_{\text{latest}}$(*oid*)) $\rightarrow (v_1, v_2, \ldots, v_N)$ get($\mathrm{M}_{\text{version}}$(*oid*, v_i)), $\forall_{i \in 1,2,\ldots,N}$ *merge*$(v_1, v_2, \ldots, v_N)$
$\text{get}_{\text{version}}$(*oid*, *version*)	get($\mathrm{M}_{\text{versions}}$(*oid*, *version*))
$\text{get}_{\text{latest}}$(*oid*)	get($\mathrm{M}_{\text{latest}}$(*oid*)) $\rightarrow (v_1, v_2, \ldots, v_N)$ get($\mathrm{M}_{\text{version}}$(*oid*, v_i)), $\forall_{i \in 1,2,\ldots,N}$

Table 5.1. Mapping of client operations to data store operations. The function *latest* outputs the subset of versions that are not older than any of those specified in the input set.

version IDs (considering a scenario with low average number of versions per object). For example, one million objects with an average of two versions each, with each version being encoded in 10 bytes on average, combined would take less than 20MB of space. Thus, caching would allow us to satisfy the client requests while removing one roundtrip to the data store.

5.4 Previous Work

Felber et al. [37] consider the problem of providing versioning to distributed strongly-consistent key-value stores. They concluded that a plain key-value store cannot emulate a versioned key-value store. However, VersionD provides versioning support on top of a non-distributed key-value store, by adding the replication layer above the versioning layer. Therefore, their *Separation Result* [37] does not hold. We are able to bring versioning support to a replicated data store through separation of concerns and reordering the software layers, effectively bypassing their limitation.

Chapter 6

Shared-Data Architecture

6.1 Architecture Overview

In this chapter we explore a specific use-case in a non-conventional architecture, by deploying GeoD on top of SindexD, Scality's [11] bucket-key-value store featuring prefix queries. The goal of deploying GeoD is to optimistically replicate the external state of SindexD. SindexD does not persist any of the data itself – it offloads all of it into a set of two DHTs, using them as logically-independent key-value stores. The key details of this architecture are:

Shared Data DHT The DHT where the object data is stored is shared between the SindexD instances in all sites. All sites can write to any location in this DHT. This DHT is not replicated – it is *stretched* among all sites. All the data placed here is immutable – if one chunk needs to be changed, the value of the new chunk is placed at a different key location in the DHT.

Private Metadata DHT The DHT where all the metadata is stored is private and only accessible by the local SindexD instance. This DHT is replicated at all the sites, i.e. each site maintains an independent copy that is only locally accessible.

Chunked Object Store Large objects are split into several fixed-size chunks, each of them mapped into a different key in the data DHT.

Chunk Indexing The data chunks are indexed via a B-Tree-like structure. The tree nodes are stored at random locations in the metadata DHT.

A pure GeoD solution is not ideal for deployment in the RING, Scality's software-defined storage system. Due to the non-deterministic placement of data in the shared ring, together with the lack of synchronization between the SindexD replicas, each site will place in the shared RING its own copy of each object. In addition, collisions may occur, as both sites may generate the same hash for different objects.

The ideal case would be for the different SindexD instances eventually map to the same locations in the shared data ring, avoiding writing and storing one copy

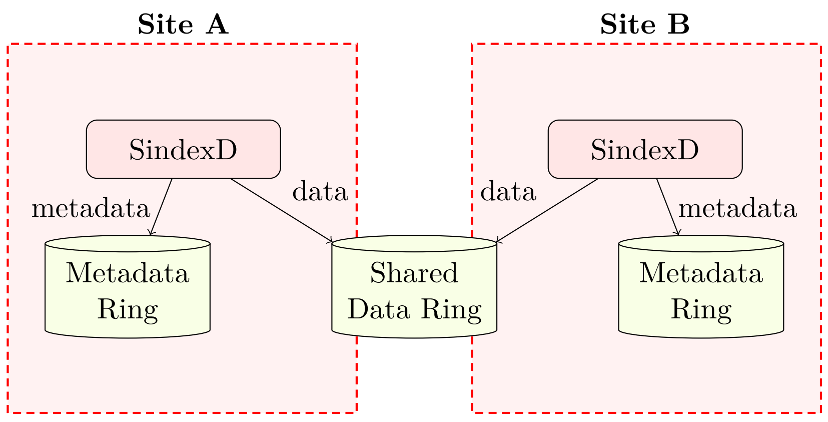


Figure 6.1. SindexD Data and Metadata Representation and Placement. Although the picture shows two sites, the actual deployment can encompass an arbitrary number of sites.

per site. However, GeoD only guarantees that the instances converge in a logical way (i.e. their apparent external state is the same, but their actual internal state may differ).

Since the internal mapping algorithms are not deterministic, the internal mapping to the items in the data ring will diverge and cause to have one copy stored per site (not including the actual replication factor for objects). To solve the mapping problem, we need to make the mapping deterministic. In addition to the mapping, GeoD also transfers all the data through the network, although it is available to read directly from the RING. This causes a significant overhead in network utilization.

6.2 Single Copy Per Chunk

Currently each object is split into several chunks, which are then indexed using a B-Tree-like structure. The nodes of the tree are stored in a private metadata ring, and all the records (the actual *data nodes*) are stored in the shared data ring. All the nodes of the B-Tree are given random identifiers for load balancing purposes. In addition, each of the records is also mapped into a random location in the ring for the same reason.

6.2.1 Deterministic Mapping

In order to make the mapping deterministic and distributed, without sacrificing the load balancing properties, we propose the utilization of consistent hashing [47]. It has all the desired properties, and allows for convergent mapping of all the replicas into the same locations.

Definition 6.2.1. Deterministic collision-free mapping.

The deterministic mapping of chunks into their location in the shared data ring is given by:

$\mathit{key}_{RING} \leftarrow \mathbb{H}(\mathit{site}, \mathit{version}, \mathit{key}, \mathit{chunk})$

If two chunks have a different number, belong to a different object, to a different version of an object, or are written by a different site, we guarantee them to have different keys (assuming no hash collisions).

By replacing the current mapping function with consistent hashing, each chunk of each object will be mapped into a specific place. In addition, by including the site ID, we guarantee that two different sites writing concurrently to the same chunk of the same object will not overwrite each others' changes.

6.2.2 Remote Update Signaling

In addition to the mapping, SindexD must now be able to distinguish between local updates and remote updates. We propose the addition of an API call, signaling that a given update is remote.

Definition 6.2.2. Additional SindexD API call to support remote update signaling:
$\mathit{addRemote}(siteID, versionID, objectID, summary)$

Where *summary* is a the list of chunks affected, or simply the number of chunks if no copy-on-write is available. No need to transfer the actual data, as every SindexD instance can simply pull it directly from the RING.

6.2.3 Traffic Duplication

The proposed solution for *data multiplication* also addresses the problem of traffic duplication, as only a summary with the IDs of the chunks affected is transmitted between the GeoD instances.

In summary, this architecture minimizes the propagation latency of updates, as well as the amount of traffic required for propagation due to the separation of data and metadata. The shared data repository may be queried on-demand to fetch the required data, instead of data being constantly broadcasted by all replicas.

The following chapter explains how to augment GeoD's functionality by adding versioning to objects, allowing for complex merge semantics and logging the values of an object.

Chapter 7

Conclusion

This work shows it is possible to extend the feature set of existing key-value stores by providing optimistic replication as a shim layer.

In this thesis we presented GeoD, a shim providing optimistic replication to independent key-value store instances. GeoD works by intercepting requests from clients to the datastore, detecting the resulting internal state transitions and propagating them to the other sites. It's built-in conflict resolution ensures that concurrent operations don't result in data loss.

The built-in design for conflict resolution follows two principles: it should be as familiar to the user as possible (i.e. it should behave similarly as if it was a non-replicated system), and should prevent unwanted loss of information – any deletion that is concurrent with a write to the same object will be ignored. Causality is tracked by resorting to version vectors. In case two values are written concurrently, the one with the largest physical timestamp wins (last-write-wins policy).

We also presented VersionD, an upgrade to GeoD which extends the client API with a versioning API, allowing for assigning multiple values to a single object. This allows for applications to use their own conflict resolution semantics, rather than being limited by the built-in mechanism. The API changes are backwards compatible – any application working with GeoD works the same with VersionD. The downside to VersionD is that an API call from a client can translate to several calls to the key-value store, increasing the latency for requests and the load on the system.

Finally, we also describe an efficient architecture for optimistically replicated data stores, using two distinct storage components – a common data repository, shared by all the sites, and a private metadata repository, optimistically replicated at all sites. This allows to reduce massively the size of synchronization messages, thus reducing bandwidth costs and propagation latency and, consequently, the number of conflicting operations.

Bibliography

[1] Amazon Simple Storage Service. `http://aws.amazon.com/s3/`. Accessed: 2015-11-12.

[2] Apache Accumulo. `https://accumulo.apache.org`. Accessed: 2015-11-12.

[3] Apache HBase. `http://hbase.apache.org`. Accessed: 2015-11-12.

[4] Google. `http://www.google.com`. Accessed: 2015-11-12.

[5] LinkedIn. `http://www.linkedin.com/`. Accessed: 2015-11-12.

[6] National Security Agency. `https://www.nsa.gov`. Accessed: 2015-11-12.

[7] Netflix. `http://www.netflix.com`. Accessed: 2015-11-12.

[8] Netflix Dynomite. `https://github.com/Netflix/dynomite`. Accessed: 2015-11-12.

[9] Project Voldemort. `http://www.project-voldemort.com/`. Accessed: 2015-11-12.

[10] Riak. `http://basho.com/riak/`. Accessed: 2015-11-12.

[11] Scality. `http://www.scality.com`. Accessed: 2015-11-12.

[12] Scality RING. `http://www.scality.com/ring/`. Accessed: 2015-11-12.

[13] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and P.W. Hutto. Causal memory: Definitions, implementation and programming, 1994.

[14] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In Rachid Guerraoui, editor, *Proceedings of DISC 2004: 18th International Symposium on Distributed Computing*, number 3274 in LNCS, pages 102–116. Springer Verlag, 2004.

[15] Paulo Sergio Almeida, Carlos Baquero, and Victor Fonte. Version Stamps - Decentralized Version Vectors. In *Proc. of the 22nd International Conference on Distributed Computing Systems*, 2002.

[16] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval Tree Clocks. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 259–274, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Paulo Sérgio Almeida, Carlos Baquero Moreno, Ricardo Gonçalves, Nuno Preguiça, and Vitor Fonte. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. In *Distributed Applications and Interoperable Systems*, volume 8460, Berlin, Germany, June 2014. Springer, Springer.

[18] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation. *CoRR*, abs/1410.2803, 2014.

[19] Artur Andrzejak and Zhichen Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of the Second International Conference on Peer-to-Peer Computing*, P2P '02, pages 33–, Washington, DC, USA, 2002. IEEE Computer Society.

[20] Hagit Attiya and Jennifer L. Welch. Sequential Consistency Versus Linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[21] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. HAT, Not CAP: Towards Highly Available Transactions. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 24–24, Berkeley, CA, USA, 2013. USENIX Association.

[22] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[23] Roberto Baldoni and Matthias Klusch. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online*, 3(2):–, February 2002.

[24] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[25] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[26] James M. Bloom and Kevin J. Dunlap. Experiences Implementing BIND, a Distributed Name Server for the DARPA Internet. In *USENIX Summer*, pages 172–181. USENIX Association, 1986.

[27] Eric A. Brewer. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[28] Alexandre Filipe Alves Tobias Campos. HBase++ Extending HBase with client-centric consistency guarantees for Geo-Replication. Master's thesis, Instituto Superior Técnico, 2015.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[30] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, August 1984.

[31] Bernadette Charron-Bost. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[32] Valter Balegas de Sousa. Key-CRDT stores. Master's thesis, Universidade Nova de Lisboa, 2012.

[33] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[34] C. Diot, W. Dabbous, and J. Crowcroft. Multipoint communication: a survey of protocols, functions, and mechanisms. *Selected Areas in Communications, IEEE Journal on*, 15(3):277–290, Apr 1997.

[35] Dropbox. `http://www.dropbox.com`. Accessed: 2015-11-12.

[36] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[37] Pascal Felber, Marcelo Pasin, Etienne Rivière, Valerio Schiavoni, Pierre Sutra, Fábio Coelho, Miguel Matos, Rui Oliveira, and Ricardo Vilaça. On the Support of Versioning in Distributed Key-Value Stores. In *33rd IEEE International Symposium on Reliable Distributed Systems - SRDS*, Nara, Japan, October 2014.

[38] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[39] Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[41] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[42] Rachid Guerraoui and André Schiper. Fault-Tolerance by Replication in Distributed Systems. In *In proc. conference on reliable software technologies*, pages 38–57. Springer Verlag, 1996.

[43] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[44] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[45] Simon Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, RFC Editor, October 2006. `http://www.rfc-editor.org/rfc/rfc4648.txt`.

[46] Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Trans. Software Eng.*, 9(3):240–247, 1983.

[47] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel 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, pages 654–663, New York, NY, USA, 1997. ACM.

[48] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube Approach to the Reconciliation of Divergent Replicas. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 210–218, New York, NY, USA, 2001. ACM.

[49] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[50] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[51] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[52] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[53] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[54] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[55] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[56] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual Consistency. *Commun. ACM*, 57(5):61–68, May 2014.

[57] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, Availability, Convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

[58] Dahlia Malkhi and Doug Terry. Concise Version Vectors in WinFS. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 339–353, Berlin, Heidelberg, 2005. Springer-Verlag.

[59] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[60] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):17–25, January 1990.

[61] Brad T. Moore and Paolo A. G. Sivilotti. Plausible Clocks with Bounded Inaccuracy. In *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 214–228, Berlin, Heidelberg, 2005. Springer-Verlag.

[62] Steve Muir. The Seven Deadly Sins of Distributed Systems. In *First USENIX Workshop on Real, Large Distributed Systems, WORLDS'04, San Francisco, CA, USA, December 5, 2004*, 2004.

[63] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of NOSQL Databases and its Comparison with Relational Databases.

[64] Jakob Nielsen. Usability Heuristics. In *Usability Engineering*, chapter 5.

[65] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, October 1979.

[66] Fernando Pedone. Boosting System Performance with Optimistic Distributed Protocols. *Computer*, 34(12):80–86, December 2001.

[67] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[68] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted Version Vectors: Logical Clocks for Optimistic Replication. *CoRR*, abs/1011.5808, 2010.

[69] Sriram Ramabhadran, Joseph Hellerstein, Sylvia Ratnasamy, and Scott Shenker. Prefix Hash Tree - An Indexing Data Structure over Distributed Hash Tables.

[70] Norman Ramsey and Elöd Csirmaz. An Algebraic Approach to File Synchronization. *SIGSOFT Softw. Eng. Notes*, 26(5):175–185, September 2001.

[71] Yasushi Saito and Marc Shapiro. Replication: Optimistic Approaches. Technical report, HP Laboratories Palo Alto, 2002.

[72] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[73] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distrib. Comput.*, 7(3):149–174, March 1994.

[74] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, January 2011.

[75] Justin Sheehy. There is No Now. *Commun. ACM*, 58(5):36–41, April 2015.

[76] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[77] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari 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, pages 149–160, New York, NY, USA, 2001. ACM.

[78] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[79] Vinh Tao, Marc Shapiro, and Vianney Rancurel. Merging Semantics for Conflict Updates in Geo-Distributed File Systems. Haifa, Israel, May 2015.

[80] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[81] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Conit-based Continuous Consistency Model for Replicated Services. *ACM Trans. Comput. Syst.*, 20(3):239–282, August 2002.

[82] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno M. Preguiça. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. *CoRR*, abs/1310.3107, 2013.