LIGHTKONE

Lightweight computation for networks at the edge

Project no.	732505
Project acronym:	LightKone
Project title:	*Lightweight computation for networks at the edge*

D6.1: New concepts for heavy edge computing

Deliverable no.:	D6.1
Title:	New concepts for heavy edge computing
Due date of deliverable:	December 31, 2017
Actual submission date:	February 7, 2018
Lead contributor:	UNIKL
Revision:	0.1
Dissemination level:	PU
Start date of project:	January 1, 2017
Duration:	36 months

This project has received funding from the H2020 Programme of the European Union

Revision Information:

Date	Ver	Change	Responsible
31/01/2018	0.1	1st version	UNIKL

Contributors:

Contributor	Institution
Annette Bieniusa	TUKL
Deepthi Akkoorath	TUKL
Peter Zeller	TUKL
Nuno Preguiça	NOVA
Gonçalo Cabrita	NOVA
Gonçalo Tomás	NOVA
João Leitão	NOVA
Bernardo Ferreira	NOVA
Dimitrios Vasilas	Scality
Roger Pueyo Centelles	UPC
Ilyas Toumlilt	UPMC/INRIA
Paolo Viotti	UPMC/INRIA

Contents

1 Executive summary

Despite their distributed nature, many edge applications rely on a cloud database for persistent storage, analysis, and re-distribution of data. WP6 investigates protocols and schemes for integrating cloud databases into edge computing (*heavy edge*). To preserve availability of the system under network partitions, edge nodes typically communicate asynchronously with the cloud database. Further, cloud databases should be geo-replicated to reduce latency. This can lead to (temporary) divergence of data replicated at different nodes: at the edge and within the database.

AntidoteDB is a geo-replicated cloud database that provides transactional causal consistency for conflict-free replicated data types. These semantics are specifically targeting edge computing systems where availability under network partitions must be guaranteed. Originated from the SyncFree project, AntidoteDB forms the basis for the heavy-edge use cases within the Lightkone project.

The major contributions of this deliverable are extensions and techniques that have been developed to make AntidoteDB employable in the heavy-edge use cases:

- We are working on a **causally consistent client-side cache** for AntidoteDB clients.
- We provide data types that allow **non-uniform replication**. This reduces the amount of data stored at individual replicas for replicated data types such as the top-K CRDT while all queries can be answered with the locally available data.
- To protect data from unauthorized access, we have developed an **access control layer** for geo-replicated causally consistent systems and provide an implementation for AntidoteDB.
- The Antidote Query Language (AQL) is a **SQL-like interface** with support for integrity constraints such as referential integrity.

The use cases from WP2 that we want to implement in the context of WP6 span a wide selection of scenarios:

- Project partner UPC investigates into a **monitoring system for the GUIFI.net community network**.
- Project partner Scality has started to implement a **weakly-consistent datastore index** that scales to large amounts of data.
- In collaboration with the French ANR project RainbowFS, project partner INRIA / UPMC develops a **CRDT-based filesystem**, named AntidoteFS.

The development of applications on weakly-consistent data stores is error-prone. Concurrent non-synchronized access to different replicas can yield to unexpected results, even for experienced programmers. We are therefore developing tools that verify the correctness of such applications, using different proof techniques:

- **Repliss** offers a domain-specific language for modeling applications and specifying invariants; it can be either used as (interactive) verification tool or automated testing tool.

- The **CEC** tool builds on the CISE logic [30]. It verifies that the model of the application's operations is safe, communtative, and stable with respect to specified invariants.

Besides AntidoteDB and transactional causal consistency, this deliverables further comprises work on other heavy-edge systems.

- **Blotter** is a protocol for transactional data stores with non-monotonic snapshot isolation semantics that allows to maintain stronger application invariants.
- We provide a detailed analysis of the **performance for reads-dominated workloads** in geo-replicated datastores and the evaluate the tradeoff between data freshness and consistency.
- In our work on **speculative execution**, we show how systems with partial replication can substantially increase throughput and reduce latency by postponing validation of executed operations.
- MIE is a **Multimodal Indexable Encryption** framework that supports mobile applications dynamically storing, sharing, and searching multimodal data (i.e. data with multiple media formats simultaneously) in public cloud infrastructures while preserving privacy.
- Finally, we developed a middleware that extends the consistency semantics of a cloud system to the clients by providing **fine-granular use of session guarantees**.

Collaboration with other WPs Most protocols and extensions that we present in this deliverable have been implemented in the AntidoteDB framework, but they are also applicable in other heavy-edge systems. While D6.1. focuses on the challenges of integrating these techniques in AntidoteDB, deliverable D3.1. presents them in a broader context.

The work on verification tools spans both WP4 and WP6. Both tools, Repliss and CEC, target a transactional shared-object programming model that has been formalized in WP4.

The use cases that have been selected for WP6 have been proposed in WP2. Following the security analysis of D2.1., the work on multimodal indexable encryption for mobile cloud-based apps addresses the data protection requirements identified in D2.1. Further, our presentation comprises an access control model with semantics targeting weakly-consistent datastores.

As discussions within the consortium has shown, heavy- and light-edge systems are covering a continuous spectrum of distributed system design. Many insights and techniques that have been developed apply therefore for both WP5 and WP6. This includes in particular the work on partial replication and related topics such as uniform replication.

2 Results

2.1 Making AntidoteDB ready for edge computing

(a) Overview of AntidoteDB

AntidoteDB is a highly available geo-replicated key-value database. AntidoteDB provides features that help programmers to write correct applications while having the same performance and horizontal scalability as AP/NoSQL databases.

The code for AntidoteDB is available at http://github.com/SyncFree/antidote, and the documentation is available at http://antidotedb.org.

Features

- CRDTs: Conflict-free replicated data types [60] are high-level data types designed to provide sensible and deterministic semantics despite concurrent updates and partial failures. AntidoteDB supports various CRDTs such as counters, sets, maps and flags as listed in https://github.com/SyncFree/antidote_crdt.

- Horizontal scalability: Data is partitioned across several servers within a data center. AntidoteDB can execute updates and read in parallel across these partitions, thus achieving better throughput than a single machine system.

- Geo-Replication: AntidoteDB replicates data across several data centers around the globe. The reads and updates are served from the local replica, without contacting remote data centers. This asynchronous communication model provides continuous functioning even when there are failures or network partitions.

- Highly available transactions: AntidoteDB provides causal consistency and atomic multi-object operations. The transactions provide a stronger programming model by which developers can program their applications without worrying about the inconsistencies due to concurrent updates.

Architecture A data center of AntidoteDB may have more than one server to support a large database that cannot be stored in a single machine. A data center stores a full replica of the database. Each server manages multiple virtual partitions that stores a non-overlapping set of objects determined using consistent hashing. An AntidoteDB deployment consists of more than one data center located across the globe. Each data center may have a different number of servers but uses the same consistent hashing mechanism to determine the partitions.

AntidoteDB consists of the following modules which implement different functionalities.

- Persistent storage: Each partition is responsible for storing the objects owned by it persistently. The persistent storage is currently implemented as a transaction log which stores all the operations to provide fast and fault-tolerant write access and efficient management of multi-versioning for CRDT objects.

- In-memory cache: Each partition maintains an in-memory cache that stores snapshots of the objects. The read operations read from this in-memory cache instead of the persistent log. Thus the in-memory cache is also responsible for generating the requested version of the objects by applying the operations.

- Transaction manager: AntidoteDB allows atomic transactions across partitions. The transaction manager coordinates reads and updates to multiple objects stored in different partitions. The default protocol supported by AntidoteDB is Cure [3], which provides transactional causal+ consistency.

- Inter-dc replication: The updates are asynchronously replicated to other data centers by the inter-dc replication module using a publish-subscribe system implemented using ZeroMQ. Each partition publishes its updates, and the remote partitions can subscribe to them. These updates are then delivered to the transaction module, which then decides how and when to make the updates visible according to the consistency semantics provided.

- External API: AntidoteDB exposes a protocol buffer interface. This interface allows clients to communicate with AntidoteDB in a language-agnostic way.

Results AntidoteDB serves as a reference platform for implementing and testing new protocols presented in this deliverable.

To support more applications, we implemented a java client library[1] that communicates with AntidoteDB over the protocol buffer interface. The library is used by the project partners to implement their research prototypes and to implement the applications that demonstrate the applicability of weakly consistent scalable databases.

To be able to evaluate the performance of AntidoteDB, we developed a benchmarking tool called FMKe[2]. This tool was modeled after Flles Medicinkort (FMK), a subsystem of the Danish National Joint Medicine Card responsible for managing patient information at a national level, and making it available to several entities such as doctors, hospitals and pharmacies. In this domain there is a need to keep records for pharmacies, treatment facilities, patients, prescriptions, patient treatments and medical events (such as taking medicine or prognosis updates). The benchmark includes a set of high-level application operations, with each of those generating a sequence of read and write operations over entities stored in the database. The set of treatment facilities, pharmacies, patients and medical staff are assumed to exist in the database for the duration of the benchmark, so these records are populated in the data store prior to the execution of the benchmark.

There are four core entities in FMKe - patients, doctors (medical staff), pharmacies and hospitals (treatment facilities). Other records appear as relations between these entities, but current benchmark workloads are heavily focused on prescriptions and entities associated with the management of prescriptions.

As depicted in Figure 2.1, FMKe is deployed as an application server, communicating with the database system through a driver. A second component is used to generate a workload of client requests over the application servers according to a specification (i.e, a set of parameters). This architecture allows us to easily write drivers for other storage

[1] https://github.com/SyncFree/antidote-java-client

[2] The paper can be found in the Appendix of this deliverable.

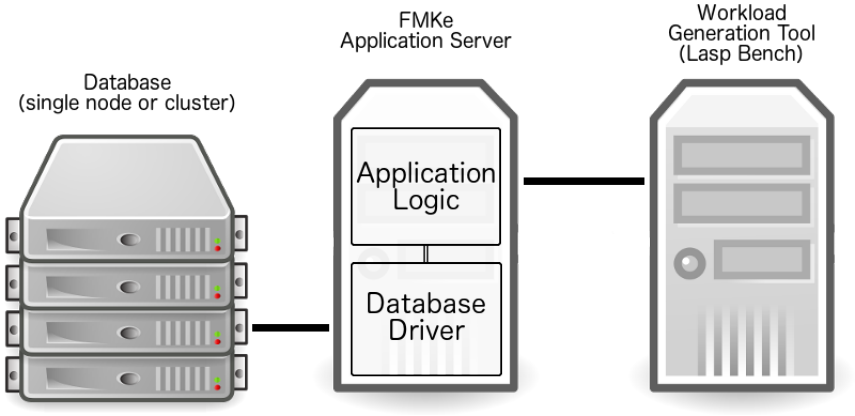


Figure 2.1: FMKe Architecture

systems and compare the performance obtained those storage systems in the context of the benchmark.

FMKe currently supports Redis, AntidoteDB, and RiakKV. We plan to extend the supported databases to include Cassandra soon. There is ongoing work to perform a comparative study of all supported data stores with different setup configurations (varying number of data centers, using in-memory storage, etc), as well as a documentation effort to further present FMKe as a standard benchmark for key-value stores for the community at large.

(b) Client-side caching

Cloud-scale services improve availability and latency by geo-replicating data in several data center across the world. Nevertheless, the closest data center is often still too far away for an optimal user experience. To remain available at all times, client-side applications need to cache data at client machines. This approach is used in many recent cloud services, where developers implement caching and buffering at application level, but it doesn't ensure system-wide consistency guarantees.

The client cache is mostly a small and size-bounded memory space, thus, it cannot contain a total replica of the data store. A common approach to solve this problem is to use partial replication [11], so each client cache contains only part of the database and its metadata.

The SwiftCloud approach Introduced by Zawirski et. al [68], the goal of the SwiftCloud approach is to extend geo-replication all the way to the client machine, pushing consistency, convergence and availability guarantees to the client cache, at a reasonable cost.

Our client cache is implemented on top of AntidoteDB (section (a)), and uses an LRU policy. Each client is connected to an AntidoteDB data center, and is interested, at any point of time, in a subset of the objects in the database, called its *interest set*. The client cache only needs to store the objects of its interest set. Initially, the client cache is a projection of data center's state, that is causally consistent. Any update, either generated by the client or delivered by the data center, maintains causal consistency. This approach

ensures that a client replica commits updates without waiting, and transfers them to its data center asynchronously.

The system guarantees the invariant that every node (DC or cache) maintains a causally-consistent set of object versions. Data is fully-replicated in a DC, and to be able to serve any version requested by the client-side cache, multiple versions of an object will be stored in the DC. On each DC, data is sharded to multiple server non-overlapping partitions, a vector clock V_P is maintained by each partition P. Any entry V_{P_i}[j] counts the number of transactions committed by P_j that P_i has processed. Each DC has a vector clock V_{DC} that maintains globally stable consistent snapshot commit time, that is the snapshot time available on all its partitions. On the client cache side, a vector clock V_C stores the most recent version of cached objects, one entry for each DC, and an additional entry for local transactions.

Each transaction in the client cache generates an identifier composed of a monotonically increasing timestamp and a unique cache identifier. A vector clock is also allocated to summarize the causal dependencies of the transaction. API functions `read` and `multi_read` returns a version of the requested object (or multiple objects for `multi_read`) that guarantees causal consistency. If the requested object is missing in the cache, it is fetched from the DC, and if its version is not valid, the `read` fails. Update operations effects are logged when an operation is executed on a previously read object, then cache's entry in V_C is updated with transaction's timestamp. The updates are made immediately visible to the client issuing them.

Each committed update at the client log is transmitted to its current DC. The client waits for an acknowledgment that contains the timestamp assigned by the DC to its update. In case of transfer failure (communication timeout or DC missing some causal dependencies) the client is switched to another DC. In the other way, client can subscribe to objects updates in the DC. In this case, the DC will maintain a FIFO best-effort channel to the client, sending a causal stream of update notifications. Those notifications contain the log updates to the objects of the client's interest set, which are then applied to its local state.

K-Stability When a DC fails, client is switched to another one. The state of the new DC may miss some client's causal dependencies. SwiftCloud's approach is to make the client cache co-responsible for the recovery of missing session causal dependencies at the new DC. We define a transaction to be K-stable at a DC, if it has been applied in at least K DCs, where K is configurable. More precisely, a client can observe the union of: (*i*) its own updates, and (*ii*) the K-stable updates made by other clients. The client can move to an other DC, as long as this new DC ensures that the client continues to observe a monotonically-growing set of K-stable updates.

(c) Non-uniform Replication

Replication is a key technique in the design of efficient and reliable distributed systems. However, as the information stored in a data store grows it becomes difficult or even impossible to store all the information at every replica. A common approach to deal with this is to rely on partial replication [20, 56, 63], where each replica maintains only a part of the total system information. As a consequence, a remote replica might need to be contacted for computing the reply to some given query, which leads to high latency costs

especially in geo-replicated settings.

Non-uniform replication [16][3] is a novel approach to replication, where each replica only needs to store part of the information (like in partial replication), but where all replicas store enough information to answer every query (like in full replication). The key insight is that for some data objects, not all data is necessary for providing the result of read operations. For example, an object that keeps the top-K elements only needs to maintain those top-K elements in every replica. However, the remaining elements are necessary if a remove operation is available, as one of the elements not in the top needs to be promoted when a top element is removed.

A top-K object could be used for maintaining the leaderboard in an online game. In such a system, while the information for each user could be kept only in the edge (the devices of the user itself) and in the data center closest to the user, it is important to keep a replica of the leaderboard in every data center in order to provide low latency and availability. For this case it would also be interesting to maintain a replica of the leaderboard at the edge, allowing read and write operations to be issued to it instantly.

Advantages of non-uniform replication The main advantage of non-uniform replication is that each replica does not require all of the data to correctly respond to read operations. This allows replicas of non-uniform objects to be much smaller – reducing the storage costs for these objects. As a direct consequence, replicas of these objects do not need to propagate every update to other replicas in order to synchronize – reducing the bandwidth costs for these objects.

Applicability of the model Non-uniform replication can be applied to all objects that store some amount of data but that for responding to read operations require only a subset of the data to be available locally. In a nutshell, the subset of the data that is required to answer read operations must be propagated to all replicas while the remaining data can either not be propagated anywhere (which raises issues regarding fault-tolerance) or it can be propagated to a small number of replicas (to tolerate some number of faults).

While non-uniform replication cannot be useful for all data types, the number of use cases appears to be large enough for it to be an interesting model in practice.

Design for integration in AntidoteDB We now briefly explain how we have introduced support for non-uniform replication in AntidoteDB.

We began by implementing some interesting data types that could benefit from this replication model [29], among them the Top-K with removals. These new data types expose the same API as the data types previously supported by AntidoteDB, with some extensions in order to properly benefit from non-uniform replication inside AntidoteDB.

To support these new data types we modified AntidoteDB's IntraDC replication layer in two ways: (i) we added a mechanism for buffering operations for a short period of time, and (ii) changed the broadcasting mechanism to allow sending certain updates only to specific replicas. The first relates to the optimization of the propagation of update operations which target non-uniform CRDTs. The intuition is that the more we wait before propagating operations the more efficient the propagation will be as less operations will need to be propagated everywhere. For example, certain elements may be added to

[3] The paper can be found in the Appendix.

a Top that later turn out to not be needed everywhere due to other elements being added. The second change relates to the optimization of the delivery of updates, as some updates are not required to be present at every replica our change makes sure that they are only delivered to a subset of replicas to ensure their durability.

Besides these changes in the replication layer, some changes to AntidoteDB's materializer were also required. Specifically, we modified the materializer to ignore no-ops and to support the changes in the status of the update operations of non-uniform CRDTs. To expand a bit on the second change, it is required to ensure that update operations that previously were not required to be present at every replica (e.g. an element that was added to the Top, but was not part of the Top-K elements) can be propagated to all replicas when needed.

(d) Access control for weakly-consistent data stores

Information systems often store sensitive information of customers, clients and users in cloud storage facilities. To protect this information from unauthorized access, the organization running such an information system needs to define a security policy to determine who may access and/or modify which subset of the data. Security policy are typically implemented in an access control system. In general, a security policy is not immutable: new information is constantly entered into the system, and organizational changes cause time and again adaptations of the security policy. For example, in social networks, users want to be able to restrict access to their personal information when interpersonal relations change. After such a policy change, the new policy must be employed in the access control system for all operations happening afterwards. Since restarting the system for each policy change is not feasible due to availability requirements, the access control system must support dynamic changes of the policy at runtime.

For strongly consistent systems, dynamically adaptable access control is well understood. Several access control models have been proposed [25, 32, 33, 54, 55] which implicitly rely on some total ordering of the operations. In weakly-consistent systems[19, 21, 45, 51], updates are accepted at any replica and propagated asynchronously to the other replicas. These synchronization messages coming from different replicas can arrive in an arbitrary order on a node, showing often even reordering of messages from the same replica during transmission. Though there is usually a well-defined order in which the operations happen on a replica, there is no global total order of all operations issued.

As a solution for this problem, we developed a model for access control in weakly-consistent cloud-storage systems. In this setting, all replicas are trusted and operate in a secure environment, but exchange updates only asynchronously. As part of this work, we defined an abstract model for weakly consistent cloud-storage systems with access control systems [67][4].

Semantics of an access control system for weak consistency An access control system is considered correct if it enforces a security policy on all replicas. Though, for weakly consistent systems, we cannot refer to **the** security policy due to potential (temporary) divergence of replicas. Depending on the order in which events become visible, we might obtain different policies for a data access on different replicas. The goal is to

[4]The paper can be found in the Appendix of this deliverable.

disambiguate the policy for events while preserving the protective properties of a policy modification.

For the specification of our access control semantics, we follow therefore two principles: (1) All policy changes need to be applied on all replicas before applying any subsequent data events and (2) conflicts of concurrent policy modifications are handled conservatively.

As we haven shown in [67], it is possible to provide these semantics by tracking of the causality regarding policy modifications and data operations, and implementing the security policies as a Policy CRDT. This CRDT offers two operations: The `setRight` operation allows to overwrite rights assignments that happened before. Specifically, it means that all rights assignments which are causally related are removed while concurrent assignments are retained before adding the new assignment to the policy. This possibly results in multiple concurrent right assignments. The `getOp` operation yields then the current right assignment by taking the most restrictive of all concurrent rights assignments.

Implementation As proof of concept, we implemented an access control layer for AntidoteDB, named ACGreGate, with these weakly-consistent semantics as an extension to this AntidoteDB Java client library. Further, we designed and implemented a Policy CRDT and included it in AntidoteDB's CRDT library.

To implement the online check, ACGreGate intercepts all operations sent by the client library to the AntidoteDB database and processes them by an access control monitor (Figure 2.2). This monitor takes an implementation of a decision procedure to make the access control decisions. The decision procedure is application-dependent and corresponds to the security policy of the application. In principle, the decision procedure takes the operation, the currently acting user, and the permissions of this user and decides whether the operation is allowed to be executed or not.

Evaluation ACGreGate has been evaluated in the context of a case study with a student management system. ACGreGate's performance is an order of magnitude better than a centralized access control server when simulating a roundtrip time of 10 ms between access control server and datastore replica. This shows that geo-replicated databases can substantially benefit from a replicated weakly-consistent access control layer.

(e) Antidote Query Language

Antidote Query Language (AQL) is a module providing a SQL-like interface for AntidoteDB. The goal of AQL is to allow application developers to continue using the familiar SQL interface for interacting with AntidoteDB. Other NoSQL databases offer SQL-like interface, such as CQL for Cassandra. AQL shares some properties with these systems. Notably, it does not provide strong consistency, allowing uncoordinated concurrent updates to occur. AQL builds on the semantics on the CRDTs available in AntidoteDB to handle these concurrent updates as detailed later, guaranteeing that all replicas converge to the same value.

Unlike other SQL-like interfaces, AQL design focuses on exploring AntidoteDB unique features for providing a semantics closer to SQL strong consistency while running under weak consistency. In this context, AQL is the first to provide support for

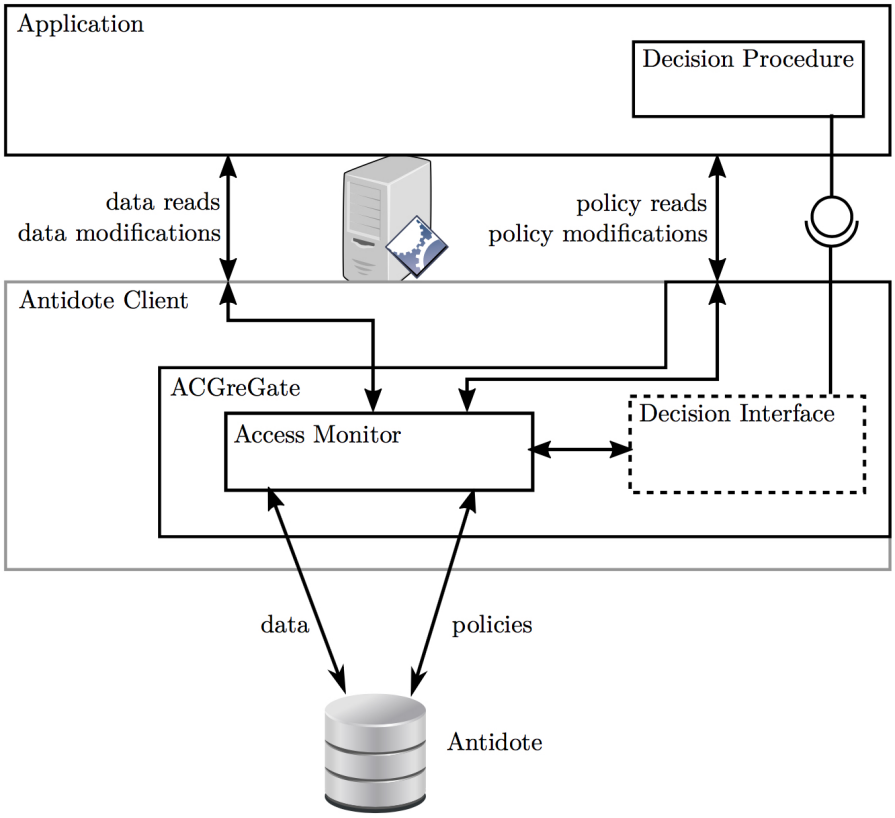


Figure 2.2: Architecture of ACGreGate.

maintaining SQL integrity constraints, including primary key, check and referential integrity constraints.

We now describe the current version of AQL, which is still under development.

Tables and DDL statements AQL supports the relational data model, where data is stored in tables with a given schema. A table is created by executing a `create table` statement with the following syntax:

```
CREATE [@AW | @RW] TABLE table_name (
  column1 datatype [constraint],
  column2 datatype [constraint],
  ...
  column_n datatype [constraint]
)
```

A table has an associated concurrency semantics: *add-wins (@AW)* or *remove-wins (@RW)* (by default, add-wins). When the *add-wins* semantics is selected, if a delete for a given primary key executes concurrently with an insert or update of a row with that primary key, the delete has no effect and the row will be kept in the table. When the *remove-wins* semantics is selected, the delete takes precedence over concurrent inserts or updates.

The `create table` statement also specifies the columns of the table, including their name and type. Currently, AQL supports the following data types with its associated concurrency semantics:

VARCHAR A `VARCHAR` field stores a string with any dimension. On concurrent updates, a *last-writer-wins* policy is adopted, by preserving the last value written (more precisely, the value of the write operation with the largest timestamp). This is achieved by mapping a `VARCHAR` column to a *last-writer-wins* register;

BOOLEAN A `BOOLEAN` field stores a boolean value. On concurrent updates, a *enable-wins* policy is adopted, by setting the value to true if at least one of the concurrent updates set the field to true. This is achieved by mapping a `BOOLEAN` column to a *enable-wins* flag;

INTEGER A `INTEGER` field stores an integer. A column with this data type can be updated by either assigning a new value or by incrementing/decrementing a constant value. For a given set of operations, the value of the field is $mv + sum_i - sum_d$, with: mv, the largest value assigned to the field such that there is no other assign operation that happens after that assignment (or 0 if no assign has been executed); sum_i, the sum of all increment operations that have not happened before the assign of mv; dec_i, the sum of all decrement operations that have not happened before the assign of mv. This is achieved by mapping a `INTEGER` column to an Integer CRDT;

COUNTER_INT A `COUNT_INT` field stores an integer with an associated check constraint. A column with this data type can be updated by incrementing/decrementing a constant value. An update operation may fail locally if its execution can violate the defined check constraint. This happens when the local replica has not enough

reservations to guarantee that the global invariant is maintained. The value of the field is $init + sum_i - sum_d$, with: *init*, the initial value of the field (equal to the limit in the check constraint, by default); sum_i, the sum of all increment operations that have not happened before the assign of *mv*; dec_i, the sum of all decrement operations that have not happened before the assign of *mv*. This is achieved by mapping a `COUNTER_INT` column to a Bounded Counter CRDT.

Like SQL and unlike other SQL modules for NoSQL databases, AQL allows a column to have an associated constraint and enforces these constraints despite concurrent updates. Constraints can be specified using the following syntax:

```
constraint ::= PRIMARY KEY |
               CHECK [GEQ|GREATER|SEQ|SMALLER] val |
               FOREIGN KEY [@FR|@IR] REFERENCES table(column)
```

A *primary key constraint* guarantees that a row can be uniquely referenced by the value of the primary key column. There can be only a primary key column and every table must have a primary key. The uniqueness of the primary key is enforced by merging concurrent inserts with the same primary key. Columns are merged by using the concurrency semantics defined for the column.

A *check constraint* guarantees that the value of a column satisfies the specified condition for all rows in the table. The check constraint is enforced in a decentralized way. When executing an operation that updates a column with an associated check constraint, the operation may succeed or fail locally. If it succeeds, it is guaranteed that the global constraint will not be violated, despite any concurrent operations.

To achieve this, AQL relies on the Bounded Counter CRDT [9] implemented in AntidoteDB. The idea is to spilt among the replicas the difference between the current value of the column and its bound. For example, for a check constraint specifying that some column must be great or equal to zero, if the current value of the column for some row is N and there are M replicas, one could assign to each replica the right to decrement by M/N. The global constraint will not be violated if a replica accepts locally only operations that cumulatively decrement the value of the value by at most M/N. When not enough rights exist locally to guarantee the execution of the operation, the operation can fail locally (or the replica may try to get more rights from other replicas).

A *foreign key* allows a row in a table to refer to a row in another table. A database enforce referential integrity or foreign key constraints if it guarantees that no reference is broken, i.e., that all refereed rows exist. When locally executing a statement that deletes a row referred by some other row, AQL adopts the SQL cascading behavior, where the row that references the deleted row is also deleted.

AQL enforces referential integrity constraints in a decentralized way, without restricting the execution of concurrent operations, by adopting either a *force revive (@FR)* or a *ignore revive (@IR)* semantics. A referential integrity constraint can be violated if concurrently an operation inserts a row that refers to another row and another operation deletes this latter row. This leads to a broken reference. When adopting the *force revive* semantics, AQL revives the deleted row, thus healing the broken reference. When adopting the *ignore revive* semantics, AQL deletes the inserted row with the broken references. We details the AQL semantics when enforcing referential integrity later.

DML statements AQL supports the common SQL statements for inserting, updating and deleting rows in a table with the following syntax:

```
INSERT INTO table_name [(column1, column2, ... column_n)]
      VALUES (value1, value2, ... value_n)

UPDATE table_name
      SET column1 expression1 AND ... AND column_n expression_n
      WHERE condition

DELETE FROM table_name WHERE conditions
```

An `INSERT` statement inserts a new row in the database. If the value of a column is not specified, a default value is assigned (AQL currently does not support *NULL* values). As mentioned before, if two inserts with the same primary key are executed concurrently, the value of each column is the result of the merge between the inserted values.

An `UPDATE` statement updates all rows of `table_name`, known in the local replica when the `UPDATE` is submitted, that conform the specified `WHERE` condition (if no `WHERE` condition is specified, it updates all rows known locally). An `UPDATE` statement has no effect over rows concurrently inserted or updated in a way that would conform the `WHERE` condition. The effects of concurrent `UPDATE` statements that modify the same rows are merged according to the column data type.

An `DELETE` statement deletes all rows of `table_name`, known in the local replica when the `DELETE` is submitted, that conform the specified `WHERE` condition (if no `WHERE` condition is specified, it deletes all rows known locally). An `DELETE` statement has no effect over rows concurrently inserted or updated in a way that would conform the `WHERE` condition.

As mentioned before, the state of the database in the presence of concurrent insert/update and delete statements that affect the same rows is controlled by the semantics defined for the table: *add-wins* or *remove-wins*.

AQL supports querying the database through the select statement with the following syntax:

```
SELECT projections FROM table_name [WHERE conditions]
```

The result of a select is computed locally in a replica. Thus, it will not include the effects of operations executed in different replicas and that have not been propagated to the replica where the select is being executed. When compared with SQL, AQL supports only a subset of the syntax (and functionality). Notably, AQL currently does not support joins.

AQL currently has no support for transactions – each operation executes as a single AntidoteDB transaction.

Semantics of referential integrity We now detail the semantics implemented in AQL when enforcing referential integrity in the presence of concurrent updates. We start by considering the simple case, where an insert operation executes concurrently with a simple delete (i.e., a delete without cascading), leading to a referential integrity violation. Figure 2.3 exemplifies this case with two tables, Artists and Albums, where Albums has a foreign key Artist referring to the the Artists table. In the example, starting in a

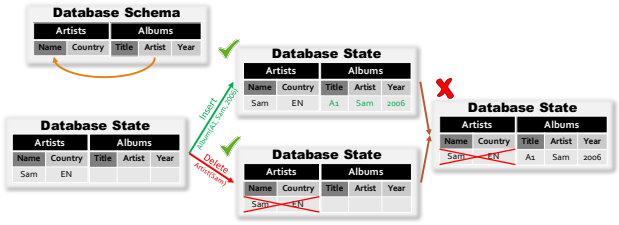
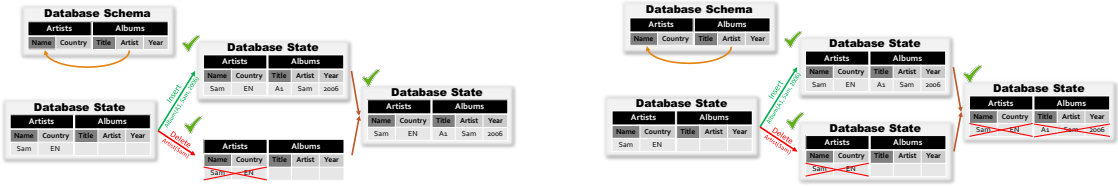


Figure 2.3: Example of integrity constraint violation.



(a) *Force revive* semantics.

(b) *Ignore revive* semantics.

Figure 2.4: Semantics for solving integrity constraint violation.

database state where only artist Sam exist, concurrently the artist is deleted and an album of the artist is inserted. When combining the effects of the two operations, we would have an album referring a deleted artist, leading to a violation of the referential integrity constraint.

As mentioned before, when adopting the *force revive* semantics, AQL revives the deleted row, thus healing the broken reference. When adopting the *ignore revive* semantics, AQL deletes the inserted row with the broken references. Figure 2.4 shows the previous example solved using these semantics.

We now consider the referential integrity violation that occurs when an insert executes concurrently with a delete with cascading behavior. Consider the example of Figure 2.5. In this example we start with a database state where there is an artist Sam with an album A0. An insert operation adds album A1 for Sam. Concurrently, a delete operation deletes artist Sam. The cascading effect leads to the deletion of album A0 (that was the only known album in the replica where the delete was executed). Combining the effects of the operations leads to a referential integrity violation with album A1 referring the deleted artist Sam.

Adopting the *ignore revive* semantics is straightforward. In such case, the inserted album A2 must be deleted also (Figure 2.6b). Adopting the *force revive* semantics should

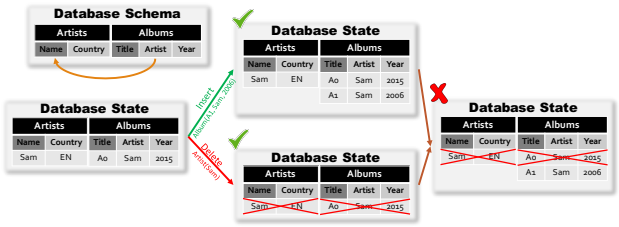
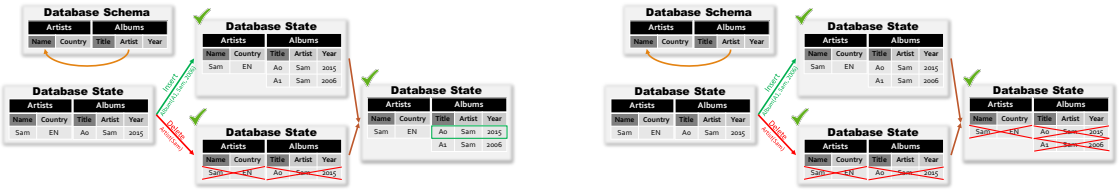


Figure 2.5: More complex example of integrity constraint violation.



(a) *Force revive* semantics.

(b) *Ignore revive* semantics.

Figure 2.6: Semantics for solving integrity constraint violation in the more complex example.

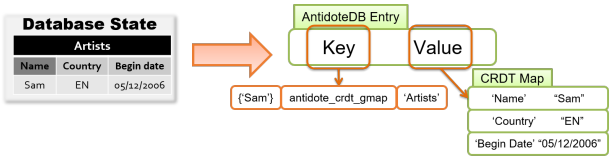


Figure 2.7: Mapping between an AQL row to an AntidoteDB key/value pair.

revive artist Sam. An interesting question in this case is what should be the effect on album A0. Reviving album A0 is not necessary for enforcing referential integrity. However, as it has been deleted as a side-effect of deleting artist Sam, our decision was to revive also album A0 – Figure 2.6a exemplifies this semantics. If the album A0 had been explicitly deleted by a delete statement on table Albums, the album would have not been revived. The general rule adopted in AQL is the following: when a row *r* is revived, all rows that were deleted as a side-effect of deleting *r* are also revived.

Implementation AQL is implemented as a module that runs in a AntidoteDB deployment. Table contents are stored in AntidoteDB by storing the contents of each row as a AntidoteDB map under a key that is created using the table name and the value of the primary key – Figure 2.7 depicts the approach.

For supporting AQL operations that need to iterate over all rows of a table, AQL maintains an index of the rows of each table in an AntidoteDB set. To this end, this set is updated whenever a new row is inserted, by adding the new primary key to the set.

For controlling whether a row is deleted or not, when concurrent operations execute, the map for each row includes a visibility entry with a multi-value register CRDT. When executing an insert or update statement, the entry of every row modified is assigned the value `I`. When executing a delete statement, the entry of every row modified is assigned the value `D`. For a table with the *add-wins* semantics, a row is considered as deleted if and only if the only value of the visibility entry is `D`. For a table with the *remove-wins* semantics, a row is considered as deleted if and only if one of the values of the visibility entry is `D`. Figure 2.8 shows an example of the evolution of the database when executing a concurrent update and delete operations.

For controlling whether a row is deleted or not when affected by the rules for enforcing referential integrity, we consider two cases: without cascading and with cascading.

We first consider the case without cascading. In this case, whenever a row that has a foreign key is inserted or updated, the visibility entry of the row referenced by the

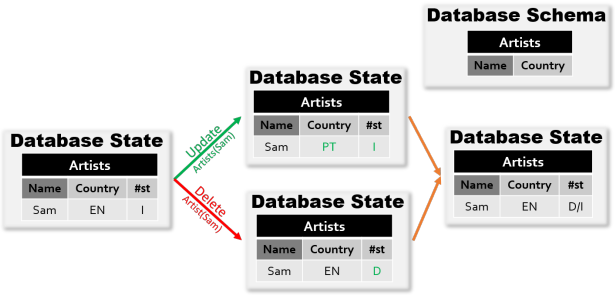


Figure 2.8: Example of the evolution of visibility entries in the presence of concurrent update and delete operations.

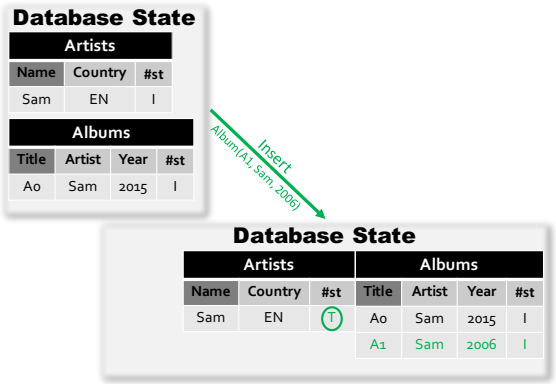


Figure 2.9: Example of the evolution of visibility flags when inserting a new row with a foreign key (without cascading).

foreign key must be set to `T`. Figure 2.9 shows how the visibility entries are updated when inserting a new Album for an existent artist.

For supporting cascading, it is not sufficient to change the visibility entries of the rows known when operations execute – for example, for implementing the semantics of Figure 2.6b, the delete of artist Sam must affect albums A0 and A1, but A1 is not known when the delete was initially executed. To address this, we record visibility tokens associated with cascading at the table level.

When deleting a row r_1 of table T_1, such that there is a row r_2 in table T_2 with a foreign key that points to r_1, besides setting the visibility entry of r_1 to `D`, AQL records at the table level of T_2 that any record pointing to r_1 has visibility token `DC`. Figure 2.10 exemplifies this case.

When inserting a row r_1 in table T_1 with a foreign key that points to row r_2 of table T_2, besides setting the visibility entry of r_1 to `I` and that of r_2 to `T`, we record at the table level of T_1 that any record pointing to r_2 has visibility token `TC`. Figure 2.11 exemplifies this case.

The visibility tokens recorded by AQL are used to decide if a given row is visible or not. To this end, for each table, AQL defines a total order among the visibility tokens depending on the semantics defined at the table level and in the foreign keys. The semantics defined at the table level defines the relative order of `I` and `D` – for *add-wins*, `D` <

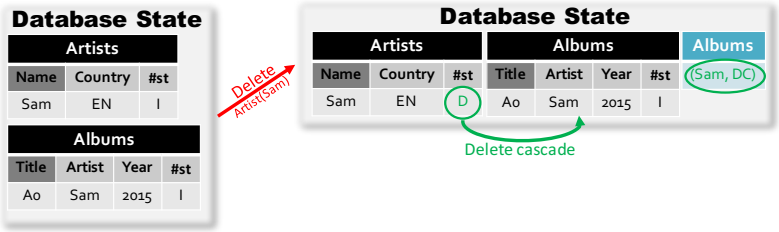


Figure 2.10: Example of the evolution of visibility entries when deleting a row (with a foreign key pointing to it).

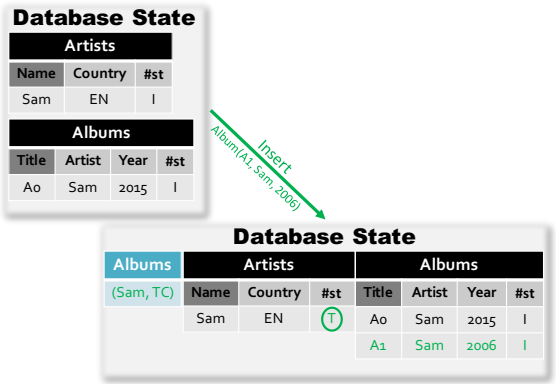


Figure 2.11: Example of the evolution of visibility entries when inserting a new row with a foreign key.

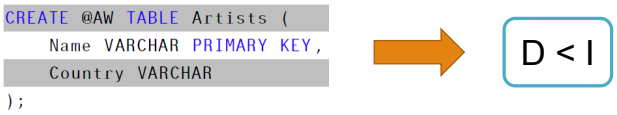


Figure 2.12: Order of visibility tokens for add-wins table semantics.

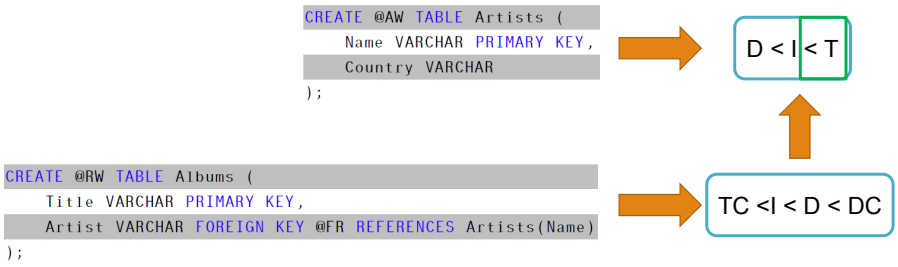


Figure 2.13: Order of visibility tokens for force-revive foreign key semantics.

`I` (Figure 2.12) and for *remove-wins*, `I < D`. The semantics defined in the foreign key defines: (i) the relation between `TC` and `DC` – for *force revive*, `DC < TC` and for *ignore revive*, `TC < DC`; (ii) the relation between `T` and the table level tokens `I` and `D` – for *force revive*, `T` is the top value (Figure 2.13) and for *ignore revive*, `T` is the bottom value.

Figure 2.14 presents the eight possible combinations, with two of them being unable to enforce referential integrity. The creation of a table that would lead to one of these invalid combination fails in AQL.

Figure 2.15 shows the visibility entries of the database for the example of Figure 2.5. With an *add-wins* semantics at the table level and a *force revive* semantics in the foreign keys, the token order in Albums is `DC < TC < D < I` and in Artists is `D < I < T`. Given these orders and the visibility tokens, all rows are visible as expected.

Final remarks AQL is the first SQL interface for a NoSQL database that enforces constraints without limiting the execution of concurrent operations. The current version of AQL is available in a branch of AntidoteDB. We are currently working on addressing some limitations of the current implementation (e.g., indices, joins) and improving the performance of the current prototype.

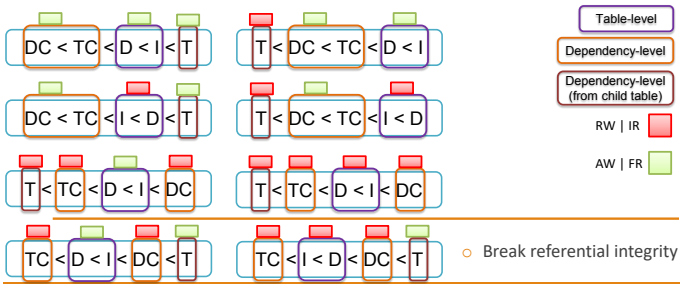


Figure 2.14: Possible orders for visibility tokens.

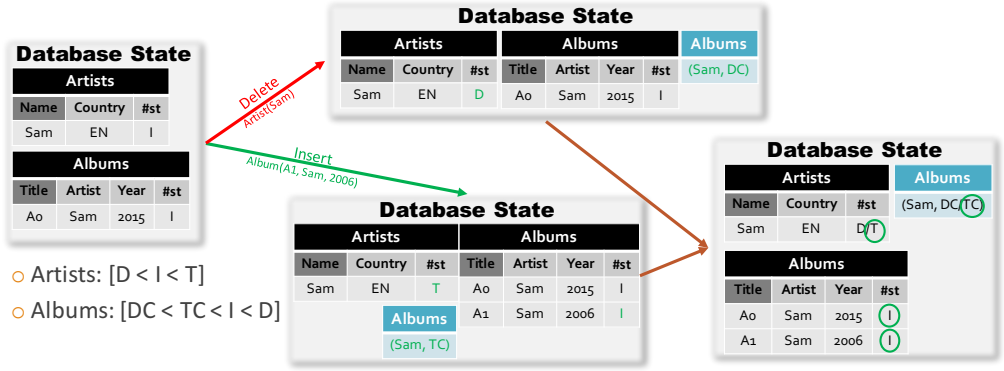


Figure 2.15: Example of Figure 2.6a with visibility entries.

2.2 Correctness and verification of heavy edge applications

(a) Repliss

Repliss is a tool supporting the development of correct applications on top of weakly consistent databases such as AntidoteDB. Such information systems often use an architecture with a separate database-access component, which is responsible for persistently storing data. When using a weakly consistent database, this is the component that has to handle the difficulties that come with eventual consistency. Even for experts, it is difficult to understand all possible interactions of possible concurrent operations and be confident in the correctness of this component.

Repliss offers a domain specific language for building the data-access component and supports developers in checking its correctness. To check the correctness, developers write the code of the application in the Repliss language and they write specifications that the code will be checked against. For checking code against specifications, Repliss supports two techniques:

1. An automatic testing tool[69][5], which executes the application code with different API invocations and concurrent schedules and checks the specification dynamically. This tool can find bugs in applications without requiring additional user inputs besides the code and specification.

2. A verification tool, which can prove that all executions adhere to the specification. The tool uses a proof technique, which was developed for Repliss, and proven sound in Isabelle/HOL[48]. The verification tool usually requires additional user input in the form of auxiliary invariants or use of an interactive theorem prover to complete the proofs, but compared to the testing tool it gives more insights and understanding and it can guarantee correctness.

Example Figure 2.16 shows an example program implementing a database of users. The function `registerUser` creates a new user account with the given data and returns the unique identifier of the newly created user. To update the mail address of a user with

[5]The paper can be found in the Appendix.

```
1  def registerUser(name: String, mail: String): UserId {
2    var u: UserId
3    atomic {
4      u = new UserId
5      call user_name_assign(u, name)
6      call user_mail_assign(u, mail)
7    }
8    return u
9  }
10
11 def updateMail(id: UserId, newMail: String) {
12   atomic {
13     if (user_exists(id)) {
14       call user_mail_assign(id, newMail)
15 }}}
16
17 def removeUser(id: UserId) {
18   call user_delete(id)
19 }
20
21 def getUser(id: UserId): getUserResult {
22   atomic {
23     if (user_exists(id)) {
24       return found(user_name_get(id), user_mail_get(id))
25     } else {
26       return notFound()
27 }}}
28
29 crdt user: Map_rw[UserId, {
30     name: Register[String],
31     mail: Register[String]
32 }]
```

Figure 2.16: Repliss Userbase Example: Procedures

a given identifier, there is a function `updateMail`. To remove a user from the system, `removeUser` can be called. The data of a user can be retrieved via `getUser`, which returns a record with the name and the email address of a given user, or `not_found` when the user does not exist.

To describe the layout of the persistent data, users can compose CRDT types from the Repliss CRDT library or specify their own datatypes using first order formulas. Choosing a suitable CRDT semantics is essential for the correctness of the application. The userbase example only works correctly if `mapDelete` affects all prior and concurrent `mapWrite` operations, so a remove-wins map CRDT can be used. Line 29 in Figure 2.16 shows how a suitable data layout for the userbase example is defined in Repliss.

For this application, we want to verify that calling `getUser(u)` after `removeUser(u)` for a user with unique identifier `u` returns `notFound`. We can specify this with the following invariant:

```
invariant (forall r: invocationId, g: invocationId, u: UserId ::
     r.info == removeUser(u)
  && g.info == getUser(u)
  && r happened before g
  ==> g.result == getUser_res(notFound()))
```

Testing Our userbase example satisfies the above property, but when we introduce a bug by removing the atomic block in the `updateMail` procedure of the userbase example, our automatic testing tool finds the counter example shown in Figure 2.17. The graph is automatically generated for the Repliss web interface. The outer boxes in the visual-

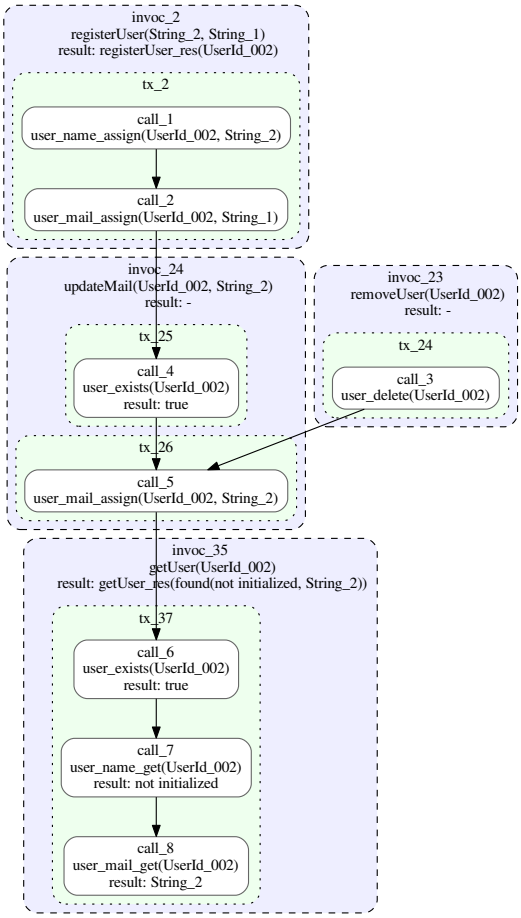


Figure 2.17: Repliss counter example with missing transaction.

izations represent invocations. Each invocation can contain several boxes representing transactions and each transaction can again contain several calls to the database. The causal dependencies between database calls are denoted by arrows. As shown in Figure 2.17, the invariant can be violated, when a concurrent `mapDelete` call becomes visible before the second database call in `updateMail`.

Verification To verify that the original program (with the transaction) really maintains the property for all possible executions, Repliss requires two auxiliary invariants:

```
invariant forall u: UserId, i: invocationId ::
  i.info == removeUser(u) && i.result != NoResult()
  ==> exists c: callId :: c.origin == i && c.op == user_delete(u)

invariant !(exists write: callId, delete: callId,
                  u: UserId, v: String ::
       (write.op == user_mail_assign(u, v) || write.op == user_name_assign(u, v))
    && delete.op == user_delete(u)
    && delete happened before write)
```

The first invariant links invocations of the data-access API with the corresponding calls happening on the database: For every completed call of the `removeUser` procedure, there must be a database call deleting the corresponding entry from the `user` map.

The second invariant states that there are no database calls updating an entry in the `user` map after it has been deleted. This invariant could be violated when we removed the atomic block, as visualized by the arrow from the `delete`-call to the `mail_assign`-call in Figure 2.17.

(b) Correct Eventual Consistency tool (CEC)

The CAP theorem states that when a network partitions (which is unavoidable in real-world scenarios), an application can either be available (AP) or consistent (CP). Most of the distributed applications lie in between the two ends of the spectrum of AP and CP. Therefore, developers have to find the optimal equilibrium between availability and consistency, one that ensures the application remains highly available while upholding the application invariants. Most of these applications use a combination of *weak* and *strong* consistency models [8, 39] to coordinate the execution of operations when the correctness of applications is at risk, and leverage the benefits of asynchronous execution when operations are safe.

In order to aid in this design process, CISE logic was introduced by Gotsman et. al [30]. The work provides a modular proof rule for verifying whether a particular consistency level for each operation preserves the application invariant. Informally, the proof rule defines the conditions (commutativity, safety, and stability) under which the application invariant is preserved.

The CEC tool [44] receives the specification of an application, and outputs pairs of operations that might break the correctness of the application if executed concurrently. With that information, it is possible to derive sets of tokens, that can be associated with operations, to pin-point where in an application the coordination is required. The CEC tool automates the proof rule defined and proved sound in [30], ensuring that the coordination generated for a given application is correct.

The specifications are written in Boogie [10], a versatile intermediate verification language, which gives the programmer the ability to specify more complex behaviors for

```
type Tournament;
type Player;
var enrollment: [Player, Tournament] bool;
var tournaments: [Tournament] bool;
var players: [Player] bool;

function invariant() returns(bool)
{
   forall t: Tournament, p: Player ::
     enrollment[p,t] ==> tournaments[t] && players[p]
}

procedure addTournament(t1: Tournament)
modifies tournaments;
requires true;
ensures forall t: Tournament ::
  t == t1 ==> tournaments[t1] == true
  &&
  t != t1 ==> tournaments[t] == old(tournaments)[t]; { }

procedure remTournament(t1: Tournament)
modifies tournaments;
requires !exists p: Player :: enrollment[p, t1];
ensures forall t: Tournament ::
  t == t1 ==> tournaments[t1] == false
  &&
  t != t1 ==> tournaments[t] == old(tournaments)[t]; { }

// addPlayer and remPlayer can be similarly defined.

procedure enroll(p1: Player, t1: Tournament)
modifies enrollment;
requires players[p1] == true && tournaments[t1] == true;
ensures forall p: Player, t: Tournament ::
  p == p1 && t == t1 ==> enrollment[p1,t1] == true
  &&
  p != p1 || t != t1 ==> enrollment[p,t] == old(enrollment)[p,t]; { }
```

Figure 2.18: Tournament management application: Boogie specification.

operations. The tool provides support for using complex data types and it is possible to write modular specifications using libraries.

System model We assume a database system composed by a set of objects fully replicated. An application is defined as a set operations and a set of invariants expressing the data integrity constraints. Each operation has an associated precondition stating the conditions that have to be guaranteed for its safe execution. When an application submits an operation to the local replica, the precondition is checked on the local database state. If the precondition holds, the operation is executed locally, and its effects are propagate asynchronously to remote replicas. Otherwise, the operation has no effect. As in [30], it assumes causal propagation of operation effects.

The abstract coordination mechanism is a *token system* as defined in [30]. It consists of a set of tokens and a symmetric conflict relation over tokens. Each operation may have an associated set of tokens, ensuring that other operations with conflicting tokens cannot be executed concurrently and their execution has to be coordinated.

It is assumed that the programmer annotates the application code with a Boogie specification that describes the database state, data invariants, preconditions, and effects of each operation. To illustrate the analysis, consider the distributed tournament management application of Figure 2.18: *addTournament*(t) and *remTournament*(t) register and

```
Operations' tokens:
enroll(p,t)      : { token_ep(p), token_et(t) }
remTournament(t) : { token_rt(t) }
addTournament(t) : { }
remPlayer(p)     : { token_rp(p) }
addPlayer(p)     : { }

Conflict relation:
token_ep(p)  : token_rp(p)
token_rt(t)  : token_et(t)
token_rp(p)  : token_ep(p)
token_et(t)  : token_rt(t)
```

Figure 2.19: Tournament management application: Generated token system.

remove tournament t, respectively; *addPlayer*(p) registers player p; and, *enroll*(p,t) enrolls player p in tournament t. This application has to ensure an integrity invariant: if player p is enrolled in tournament t, both player p and tournament t must be registered.

The input specification is then analysed in three distinct steps.

Safety analysis The first step checks whether each individual operation preserves the invariant. This is done to validate the correction of the specification given as input. In the example, if operation *remTournament*(t) did not have the precondition requiring that no player is enrolled in tournament t, the operation would fail the safety analysis.

Commutativity analysis The second step verifies commutativity between all pairs of operations. It outputs the subset of these pairs that are not commutative, as well as the sets of tokens needed to address this issue. In the example, operations *addTournament*(t) and *remTournament*(t) do not commute, while *addPlayer*(p) and *addTournament*(t) are commutative.

Stability analysis The third step checks the stability of each operation precondition against all other operations effects. It provides the programmer with the set of pairs of operations that cannot be executed concurrently, as they can break the application's invariant. It also outputs the set of tokens needed to avoid their concurrent execution. For example, the precondition of *enroll*(p,t) is not stable under concurrent execution of *remTournament*(t), while the precondition of *addTournament*(t) is stable under the effects of *enroll*(p,t).

In each step of the analysis the tool generates a set of tokens that are used to prevent conflicting pairs of operations from executing concurrently. These tokens are based on the parameters of each operation. More specifically, the tool checks different parameter values for each pair of operations, identifying which combinations of parameters might invalidate the invariant. The tool leverages the Boogie verification engine to detect efficiently the problematic combination of parameters. The output is a token system, indicating the relations between conflicting parameters (see Figure 2.19).

Previous work [8, 30] has demonstrated that replicated data types (CRDTs) can be used to solve some conflicting pairs of operations without using coordination. CEC tool provides a small library of generic CRDT types that can be used by the programmer. With this library the programmer has the choice between using either tokens or CRDTs, as a way to solve conflicting operations.

An experience report is available [47]. The report demonstrates various sample specifications verified using the tool and the recommendations on improving it.

2.3 Further work on data storages for heavy-edge systems

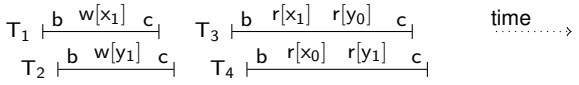
(a) Blotter: Geo-replication with strong consistency

AntidoteDB is a geo-replicated storage systems that adopts a weak consistency model for providing low latency and high availability. However, there is a class of applications that cannot run under weak consistency, and require strong consistency and transactions. For instance, many applications within Google are operating on top of Megastore [7] and Spanner [18], systems that provides ACID semantics within the same shard or in the global database. A number of other works have proposed protocols for providing strongly consistent geo-replication [36, 42, 43, 53, 61, 70].

Besides exploring the Just-Right Consistency approach, and providing mechanisms that allow applications to maintain global invariants while minimizing coordination, we are also exploring alternative protocols for run transactions with strong consistency for geo-replicated data. In this section we overview Blotter [46][6], a protocol for executing transactions in geo-replicated storage systems with non-monotonic snapshot isolation semantics. For the readers convenience the complete text of the original publication is reproduced in the Appendix.

Non-monotonic snapshot isolation NMSI is an evolution of Snapshot Isolation (SI). Under SI, a transaction (logically) executes in a database snapshot taken at the transaction begin time, reflecting the writes of all transactions that committed before that instant. Reads and writes execute against this snapshot and, at commit time, a transaction can commit if there are no write-write conflicts with concurrent transactions. (In this context, two transactions are concurrent if the intervals between their begin and commit times overlap.)

NMSI weakens the SI specification in two ways. First, the snapshots against which transactions execute do not have to reflect the writes of a monotonically growing set of transactions. In other words, it is possible to observe what is called a "long fork" anomaly, where there can exist two concurrent transactions t_a and t_b that commit, writing to different objects, and two other transactions that start subsequently, where one sees the effects of t_a but not t_b, and the other sees the effects of t_b but not t_a. The next figure exemplifies an execution that is admissible under NMSI but not under SI, since under SI both T_3 and T_4 would see the effects of both T_1 and T_2 because they started after the commit of T_1 and T_2.



Second, instead of forcing the snapshot to reflect a subset of the transactions that committed at the transaction begin time, NMSI gives the implementation the flexibility to reflect a more convenient set of transactions in the snapshot, possibly including

[6]The paper can be found in the Appendix.

transactions that committed *after* the transaction began. This property, also enabled by serializability, is called *forward freshness* [53].

Definition 2.1 (Non-Mon. Snapshot Isol. (NMSI)) *An implementation of a transactional system obeys NMSI if, for any trace of the system execution, there exists a partial order $\prec$ among transactions that obeys the following rules, for any pair of transactions t_i and t_j in the trace:*

1. *if t_j reads a value for object x written by t_i then $t_i \prec t_j \wedge \nexists t_k$ writing to $x : t_i \prec t_k \prec t_j$*
2. *if t_i and t_j write to the same object x then either $t_i \prec t_j$ or $t_j \prec t_i$.*

The example in Figure 2.20 obeys NMSI but not SI, as the depicted partial order meets Definition 2.1.

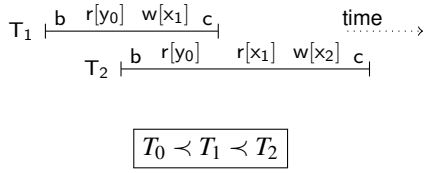


Figure 2.20: Example execution obeying NMSI but not SI. This assumes the existence of a transaction T_0 that writes the initial values for x and y.

Protocols Blotter is designed to run on top of any distributed storage system with nodes spread across one or multiple data centers. We assume that each data object is replicated at all data centers. Within each data center, data objects are replicated and partitioned across several nodes. We make no restrictions on how this intra-data center replication and partitioning takes place.

The client library of Blotter exposes an API with the expected operations: `begin` a new transaction, `read` an object given its identifier, `write` an object given its identifier and new value, and `commit` a transaction, which either returns commit or abort.

The set of protocols that comprise Blotter, detailed in appendix A, are organized into three different components:

Blotter intra-data center replication. At the lowest level, we run an intra-data center replication protocol, to mask the unreliability of individual machines within each data center. This level must provide the protocols above it with the vision of a single logical copy (per data center) of each data object and associated metadata, which remains available despite individual node crashes. We do not prescribe a specific protocol for this layer, since any of the existing protocols that meet this specification can be used.

Blotter Concurrency Control. These are the protocols that ensure transaction atomicity and NMSI isolation in a single data center, and at the same time are extensible to multiple data centers by serializing a single protocol step.

Inter-data Center Replication. This completes the protocol stack by replicating a subset of the steps of the concurrency control protocol across data centers. It implements state machine replication [37, 57] by judiciously applying Paxos [38] to the concurrency control protocol to avoid unnecessary coordination across data centers.

Final Remarks In the context of AntidoteDB we are currently studying alternatives to support applications that require strong consistency. Blotter is one those alternatives, as it allows executing transactions in geo-replicated storage systems with non-monotonic snapshot isolation semantics. When compared with alternative semantics for strong consistency, NMSI allows more concurrency and better performance, as shown in the evaluation presented in Moniz et. al. [46].

(b) Tradeoffs in reducing read latencies

This work [64] studies the costs of reading data in a distributed, transactional storage system. In particular, we try to understand whether it is possible to provide strong read guarantees while ensuring both fast performance and fresh data. Read guarantees are useful as they simplify the development of applications over distributed storage by disallowing a number of anomalies sourced in concurrency, pervasive in distributed environments. Intuitively, stronger guarantees will heavier implementations. A recent paper from Facebook (whose performance is strongly read-dominated) states: *"stronger properties have the potential to improve user experience and simplify application-level programming [... but] are provided through added communication and heavier-weight state management mechanisms, increasing latency [...] This may lead to a worse user experience, potentially resulting in a net detriment"* [2]. Is this wariness justified, i.e., is it inherently impossible to combine fast reads with strong guarantees, or can the situation be improved by better engineering? This work provides a formal and operational study of the costs and trade-offs. Our main finding is that there is a three-way trade-off between read guarantees, read delay (and hence latency), and freshness, and that some desirable combinations are impossible. Our second contribution is protocols that are optimal in terms of latency, and offer different semantics and levels of freshness.

It is well known that non-serialisable models, such as Snapshot Isolation [12] or Highly-Available Transactions [5], can improve availability and performance, particularly in highly distributed deployments such as edge environments. Therefore, this study does not necessarily assume that updates are totally ordered.[7] Furthermore, we allow weakening the read guarantees: in addition to Atomic Visibility, the strongest guarantee, which is assumed in most classical transactional models, we also consider (the weaker) Order-Consistent Visibility, enforced by recent causally-consistent systems [41], and (the weakest) Committed Visibility, equivalent to the traditional Read Committed isolation level [4]. Finally, we also consider the *freshness* dimension, because (as we show) decreasing the read delay sometimes forces to read a version of the data that is not the most recent.

Snapshot guarantees. *Snapshot guarantees* constrain the states of the data items that can be accessed by a given snapshot. The stronger guarantees provide higher isolation, and thus facilitate reasoning by the application developer. As we shall see, the weaker ones enable better performance along the freshness and delay metrics.

We distinguish the following levels:

- At the weakest level, *Committed Visibility*, a snapshot may include any updates that have been committed. As it sets no constraints between items, it allows many

[7] Enforcing a total order of transactional updates enables Consistency under Partition (CP); but, conversely, Availability under Partition (AP) requires accepting concurrent updates [59].

anomalies. For performance reasons, it is used in several production systems [15, 17, 50].

- *Order-Consistent Visibility* strengthens Committed Visibility by ensuring that the snapshot is consistent with a some (partial or total) order relation O.
  O might be the (partial) happens-before order, in order to enforce causal consistency [1, 37], or the total order of updates in the context of a strong isolation criterion such as Serialisability or Snapshot Isolation [12, 14].
- The strongest is *Atomic Visibility*, which is order-consistent, and additionally disallows the "broken reads" phenomenon [6]: if the transaction reads some data item written by another transaction, then it must observe all updates written by that transaction (unless overwritten by a later transaction).
  Many isolation levels require Atomic Visibility, e.g., Serialisability, Snapshot Isolation or Transactional Causal Consistency [3].

Order-Consistent Visibility ensures that transactions do not observe gaps in a prescribed order relation. Consider, in a social network, the data items *photos* and *acl* representing user Alice's photo album and the associated permissions. The set of their states (initially $photos_0$ and acl_0) is ordered by the happened-before relation $\rightarrow$ [37]. Alice changes the permissions of her photo album from public to private (new state acl_1), then adds private photos to the album (state $photos_2$). Thus, $acl_0 \rightarrow acl_1 \rightarrow photos_2$. Order-Consistent visibility disallows the situation where Bob would observe the old permissions (acl_0) along with the new photos ($photos_2$), missing out on the restricted permissions (acl_1).

This pattern, where the application enforces a relation between two data items by issuing updates in a particular order, is typical of security invariants [59]. It also helps to preserve referential integrity (create an object before referring to it, and destroy references before deleting the referenced object). Order-Consistent Visibility under Happened-Before order is called Causal Consistency [1].

Atomic Visibility serves to maintain equivalence or complementarity between data items [59]; for instance, ensuring that Alice is a friend of Bob if and only if Bob is a friend of Alice. Thus, it helps to maintain materialised views [6]; for instance, materialising the cardinality of a set (say the set of comments on a post) by atomically observing the updates to the set and to the materialised count.

Delay. Read latency is an important performance metric, especially for services that are heavily read-dominated, such as social networks. For instance, serving a Facebook page requires several rounds, where each round reads many items, and what is read in one round depends on the results of the previous ones; this amounts to tens of rounds and thousands of items for a single page [15]. Furthermore, serving requests with low latency keeps users engaged and directly affects revenue [22, 40, 58].

The fastest read protocol, which will be our baseline, is one that addresses multiple servers in parallel within a round, and where any one server responds immediately, in a single round-trip, without coordinating with other servers. Intuitively, this design makes it difficult to ensure strong snapshot guarantees.

We will characterise protocols by estimating the *added delay* above this baseline. *Minimal delay* is identical to the baseline. Examples include Linkedin's Espresso [50] and Facebook's Tao [15].

Bounded delay means that parallel reads are not supported, that a small number of

retry round-trips may occur to read from a server, and/or that a server may block for a small amount of time before replying to a read request. An example is COPS, which sometimes requires a second round-trip to storage servers to read a causally-consistent snapshot [41].

Mutex reads/writes means that a read might be delayed indefinitely by writes, or vice-versa, because the protocol disallows the same data item from being read and written concurrently (e.g., Google's Spanner strictly-serialisable transactions [18]).

Freshness. Another important metric is how recent is the data returned by a read. Users prefer recent data [65]; some isolation levels (e.g., Strict Serialisability) require data to be the latest version; and in others (e.g., Snapshot Isolation) serving recent data makes aborts less likely and hence improves overall throughput [49, 53]. Storing only the most recent version of a data item enables update-in-place and avoids the operational costs of managing multiple versions. In particular, keeping multiple versions of each data item on servers with restricted resources, such as edge-servers, can be problematic.

However, MVCC protocols [13] require maintaining multiple versions of a data item. Serving an old item may be faster than waiting for the newest one to become available; indeed, it would be easy to be both fast and consistent, by always returning the initial state.

Freshness is a qualitative measure of whether snapshots include recent updates or not. The most aggressive is *Latest Freshness*, which allows a server to return the most recent committed version of any data item that it stores. Intuitively, systems like Espresso and Tao [15, 50], which do not make strong snapshot guarantees, can read with no minimal delay under latest freshness.

The most conservative is *Stable Freshness*, which enables fast reads by restricting a server to return data from a *stable snapshot*, a snapshot known to be ready when the transaction started. Spanner's serialisable read-only transactions [18] exhibit this characteristic.

The intermediate level *Forward Freshness* does not necessarily return the latest version, but allows a server to read updates that are not stable, for instance those from a committed transaction that ran concurrently with the reader. For instance, COPS's causally-consistent snapshot reads [41] exhibit forward freshness.

Optimal reads. We say a protocol has optimal read performance (in short, provides optimal reads) if it ensures both minimal-delay and latest freshness. An optimal-read protocol is one that supports parallel reads, and where a server is always able to reply to a read request immediately, in a single round trip, with the latest committed version that it stores.

The trade-off. Figure 2.21 illustrates the three-way trade-off between read guarantees, read delay, and read freshness. We summarize here the main results and implications for the edge environment:

- under Atomic Visibility, it is possible to read with no extra delay (compared to a non-transactional system), but then the freshest data is not accessible, only data that was stable (written and acknowledged) before the transaction started. In practice, this can require keeping multiple object versions of each data item, which might not be an option in some edge environments.

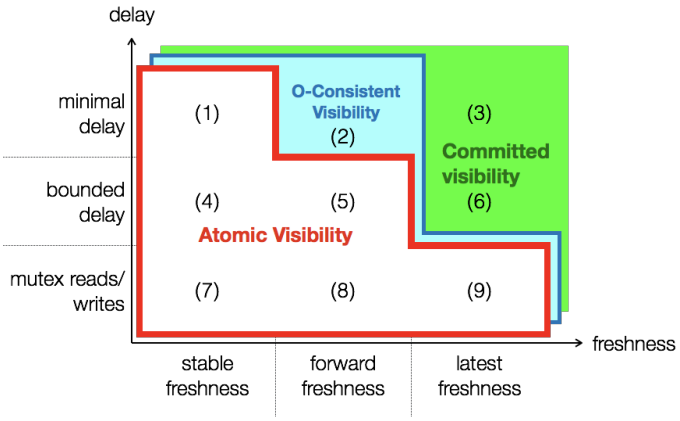


Figure 2.21: The three-way trade-off. The boxed areas represent possible guarantee/read delay/freshness combinations. Upwards and right is better performance; guarantees are stronger from back to the front planes.

- Under Order-consistent (e.g., causally-consistent) transactional reads, it is possible to read with no extra delay, while achieving optimal freshness. Indeed, this model is a contribution of this work. Such model might be interesting to provide some level of isolation to applications in the edge, at a modest cost in multi-version overhead.
- If, on the other hand, the application requires the freshest data, under either Atomic or Consistent Visibility this is possible only under a protocol where reads and writes are mutually exclusive, e.g., a read might be delayed (blocked, or in a retry loop) indefinitely by writes, or vice-versa.
- Finally, the only model that allows transactions to access the freshest data with no extra delay is Committed Visibility.

Protocols. As a practical validation of these results, we design and implement three protocol variants, all with minimal delay. We apply the trade-off analysis to protocol design. Motivated by the tight performance requirements of cloud services, we modify an existing protocol to derive three minimal-delay read protocols, called CV, OC and AV, which ensure respectively Committed Visibility, Order-Consistent Visibility, and Atomic Visibility.

We derive these protocols from Cure [3], the transactional causally-consistent protocol of AntidoteDB. Cure ensures Atomic Visibility, bounded delay and forward freshness. Using the insights taken from the analysis, we improve upon Cure's delays to ensure minimal-delay: one can either degrade read semantics or freshness. Each protocol occupies a different point in the three-way trade-off. Cure belongs in Sector 5 of Figure 2.21. Since CV ensures Committed Visibility, it can have both minimal delay and latest freshness (Sector 3 in the figure). To provide Order-Consistent Visibility with minimal delay, the best possible freshness for OC is forward freshness (Sector 2). Similarly, to provide Atomic Visibility, AV has stable freshness (Sector 1).

In our evaluation measurements in a deployment consisting of 32 machines, the three protocols have similar latency; our protocol for Committed Visibility always observes the most recent data, whereas freshness degrades negligibly for Order-Consistent reads, and the degradation is severe under Atomic Visibility.

(c) Transparent speculation in partially replicated transactional stores

Online services are often deployed over geographically-scattered data centers, which allows services to be highly available and reduces access latency. On the downside, to provide ACID transactions, global certification (i.e., across data centers) is needed to detect conflicts between concurrent transactions executing at different data centers. The global certification phase reduces throughput because transactions need to hold pre-commit locks, and it increases client-perceived latency because global certification lies in the critical path of transaction execution.

Internal and external speculation. We investigate the use of two speculative techniques to alleviate the above problems: *speculative reads* and *speculative commits*.

Speculative reads allow transactions to observe the data item versions produced by pre-committed transactions, instead of blocking until they are committed or aborted. Speculative reads can reduce the effective duration of pre-commit locks, thus increasing throughput and reducing latency. Speculative reads are a form of *internal speculation*, as misspeculations never surface to clients.

Speculative commits remove the global certification phase from the critical path of transaction execution, which further reduce user-perceived latency. Speculative commits are a form of *external speculation*, since they expose to clients the results produced by transactions still undergoing global certification. Thus, speculative commits require programmers to define compensation logic to deal explicitly with misspeculations.

Avoiding the pitfalls of speculation. Past work has shown that the use of speculative reads and speculative commits [31, 34, 52] can enhance the performance of transactional systems. However, these approaches suffer from several limitations:

1. Unfit for geo-distribution/partial replication. Some existing works in this area were not designed for partially replicated geo-distributed data stores, as they either target full replication [52] or rely on a centralized sequencer that imposes prohibitive costs in WAN environments [34].

2. Subtle concurrency anomalies. Existing geo-distributed transactional data stores that support speculative reads [31] expose applications to anomalies, e.g., data snapshots that reflect partial updates of transactions or include versions of conflicting concurrent transactions. Such anomalies can be dangerous, as they can lead applications to exhibit unexpected behaviors (e.g., crashing or hanging in infinite loops) and externalize erroneous states to clients.

3. Performance robustness. In adverse scenarios (e.g., high contention), the injudicious use of speculative techniques can severely penalize performance, rather than improving it.

To overcome the aforementioned problems, we propose Speculative Transaction Replication (STR), a novel speculative transactional protocol for partially replicated geo-distributed data stores. STR avoids the problems of centralization by using loosely synchronized clocks, similar to Clock-SI [23]. STR avoids the concurrency anomalies introduced by speculation by obeying a new concurrency criterion called Speculative Snapshot Isolation (SPSI). In addition to guaranteeing Snapshot Isolation (SI) for *committed transactions* [23], SPSI allows an *executing transaction* to read data item versions committed before it started (as in SI), and to atomically observe the effects of non-conflicting transactions that originated on the same node and pre-committed before it started. Finally, to enhance performance robustness STR employs a lightweight self-tuning mechanism that

uses hill climbing based on workload measurements to dynamically adjust the aggressiveness of the speculative mechanisms. Our evaluation shows that the use of internal speculation yields 10× throughput increase and 10× latency reduction in a fully transparent way. Furthermore, applications that exploit external speculation can achieve a reduction of user-perceived latency by up to 100×. These numbers are achieved for both synthetic and realistic workloads with low inter-data center contention, while the self-tuning mechanism ensures gradual fallback to a standard non-speculative processing mode as contention increases.

(d) Multimodal Indexable Encryption for Mobile Cloud-based Applications

Following the security analysis of D2.1., it also comes of high importance addressing data protection requirements in the heavy edge. In this context, we have been exploring secure encryption schemes and protocols that can protect data privacy and integrity while still allowing its efficient operation and computation. Efficiently indexing and searching of encrypted data in the cloud, in particular, would be very a interesting result for the project, as it would allow lightweight devices on the light edge to securely outsource their most expensive computations to the heavy edge.

Addressing this problem we proposed MIE, a Multimodal Indexable Encryption framework that supports mobile applications dynamically storing, sharing, and searching multimodal data (i.e. data with multiple media formats simultaneously) in public cloud infrastructures while preserving privacy [26][8]. MIE's functionality and security are based on a novel family of encoding algorithms with cryptographic properties, that we also proposed, called DPE - Distance Preserving Encodings. DPE schemes securely encode data while preserving a controllable distance function between plaintexts. By extracting feature-vectors from multimodal data and encoding them with DPE, mobile devices running MIE are able to outsource training and indexing computations to the cloud in a privacy-preserving way.

DPE: Distance Preserving Encodings Informally, Distance Preserving Encodings (DPE) are a family of encoding schemes that preserve a controllable distance function between plaintexts, by means of their respective encodings. We say the distance function is controllable, meaning that on instantiation of a DPE scheme a security threshold parameter should be defined, which will allow controlling the amount of information leaked by encodings. More specifically, DPE encodings should only preserve distances between plaintexts up to the value of the threshold. For greater distances, nothing should be leaked by DPE encodings. This threshold allows defining an upper bound on information leakage and security, as it will limit the adversarial ability to perform statistical attacks and establish a distance relation between different plaintexts in the application domain. More formally:

Definition 2.2 (Distance Preserving Encoding) *A Distance Preserving Encoding (DPE) scheme is a collection of three polynomial-time algorithms (*KEYGEN, ENCODE, DISTANCE*) run by a client and a server, such that:*

• $K,t \leftarrow \text{KEYGEN}(1^k)$*: is a probabilistic key generation algorithm run by the client to setup the scheme. It takes the security parameter k and returns a secret key K and a*

[8]The paper can be found in the Appendix.

distance threshold t, both function of and polynomially bounded by k.

• $e \leftarrow \text{ENCODE}(K, p)$*: is a deterministic algorithm run by the client to encode plaintext p with key K, with p polynomially bounded by k. It outputs an encoding e.*

• $D \leftarrow \text{DISTANCE}(e_1, e_2)$*: is a deterministic algorithm run by the server that takes as input two encodings* e_1 *and* e_2*. For plaintext distance function* $[0,1] \leftarrow d_p(\cdot,\cdot)$ *and encoded distance function* $[0,1] \leftarrow d_e(\cdot,\cdot)$ *(possibly* $d_p = d_e$*) with inputs polynomially bounded by k, it outputs* $D = d_e(e_1, e_2) = d_p(p_1, p_2)$*, if* $d_p(p_1, p_2) < t$*. Otherwise it outputs* $D = t$*.*

Given the definition of DPE, in [26] we formally specify all information leaked by its algorithms to an honest-but-curious cloud adversary. Additionally, two implementations of DPE are provided, one for dense media types (e.g. images, audio, and video), and another for sparse media (e.g. text). Both implementations are then used to implement an efficient prototype of MIE.

MIE: a Multimodal Indexable Encryption Framework In MIE we leverage the previous DPE definition and its two implementations to outsource training and indexing computations of multimodal data from mobile devices to the cloud servers. This is done in a privacy-preserving way by having users extract feature-vectors from the different media formats, encode them with DPE, and upload the encodings to the cloud for computation.

From a systems perspective, MIE is defined as a distributed framework with two main components: one running in the mobile device(s), which processes multimodal data, extracts feature-vectors in their different modalities, and encrypts them; and another (untrusted) running in the cloud servers, which performs training tasks and indexes data-objects through their encoded features. More formally:

Definition 2.3 (Multimodal Indexable Encryption) *A Multimodal Indexable Encryption framework is a collection of five polynomial time algorithms (*CREATEREPOSITORY, TRAIN, UPDATE, REMOVE, SEARCH*) executed collaboratively between a user and a server, such that:*

• $\mathbf{rk_R} \leftarrow \textbf{CreateRepository}(\mathbf{ID_R}, \mathbf{1^{sp_R}}, \{\mathbf{ID_{m_i}}\}_{\mathbf{i=0}}^{\mathbf{n}})$*: is an operation started by the user to initialize a new repository identified by* ID_R*. It also takes as input a security parameter* sp_R *and the n modalities to be supported by R (*$\{ID_{m_i}\}_{i=0}^{n}$*). It creates a repository representation on the server side and outputs a repository key* rk_R*.*

• $\textbf{Train}(\mathbf{ID_R}, \mathbf{rk_R}, \{\mathbf{ID_{m_i}}, \mathbf{ip_{m_i}}\}_{\mathbf{i=0}}^{\mathbf{n}})$*: operation invoked by the user to initialize repository R's indexing structures, by performing machine learning tasks (i.e. automatic training procedures), and index its data-objects, if any. The user also inputs the repository key and the indexing algorithms to be used as indexing parameters (*$\{ID_{m_i}, ip_{m_i}\}_{i=0}^{n}$*, one for each modality.*

• $\textbf{Update}(\mathbf{ID_R}, \mathbf{ID_p}, \mathbf{p}, \mathbf{dk_p}, \mathbf{rk_R}, \{\mathbf{ID_{m_i}}\}_{\mathbf{i=0}}^{\mathbf{n}})$*: is the operation used to dynamically add or update a data-object p in repository R. In addition to p, it also takes as input* ID_R *and* ID_p *(deterministic identifiers of R and p, respectively),* dk_p *(data key to be used in the encryption of p),* rk_R *(repository key of R) and* $\{ID_{m_i}\}_{i=0}^{n}$ *(the modalities represented in p).*

• $\textbf{Remove}(\mathbf{ID_R}, \mathbf{ID_p})$*: is an operation that allows a user to fully remove a data-object p from repository R and its indexing structures.*

• $\{\mathbf{ID_{p_i}}, \mathbf{p_i}, \mathbf{score^q_{p_i}}\}^{\mathbf{k}}_{\mathbf{i=0}} \leftarrow \mathbf{Search}(\mathbf{ID_R}, \mathbf{q}, \mathbf{rk_R}, \{\mathbf{ID_{m_i}}\}^{\mathbf{n}}_{\mathbf{i=0}}, \mathbf{k})$*: is issued by a user to search in repository R with object q as query, returning the k most relevant data-objects in the repository. Also takes as input the repository key* rk_R *and the modalities represented in q* ($\{ID_{m_i}\}^n_{i=0}$).

Given MIE's definition, in [26] we detail its construction and implementation. We also clearly define the leakage of MIE's operations and formally prove its security properties.

(e) Consistency Upgrades for Online Services

Online services, such as Facebook or Twitter, have public APIs to enable an easy integration of these services with applications. In a previous work [27], we have shown that these services exhibit consistency anomalies, providing a consistency levels weaker than causal consistency. This makes designing applications complex, since application developers have to reason on how to enforce the semantics of their applications in the presence of those anomalies.

To overcome this challenge, we have developed a middleware that enables a fine-grained control over the session guarantees that comprise the consistency semantics provided by such systems, without having to gain access or modify the implementation of those services. To demonstrate the feasibility of our approach we have applied it for the Facebook public API and the Redis datastore, allowing to have fine-grained control over the consistency semantics observed by clients with a small local storage and modest latency overhead.

As this work can be used in the light edge, it is also addressed in D5.1. Here, we briefly overview the algorithms proposed. This work was originally published in [28]. For the readers convenience the complete text of the original publication is in the Appendix.

Algorithms In this section we detail the algorithms that are employed by our Middleware layer to enforce session guarantees, and the rationale for their design. To this end, we briefly remind what each of the four session guarantees entails (we extend the definitions previously introduced in [28]), and then explain why our algorithms ensure that the anomalies associated with each of the session guarantees are prevented by it.

We explain our algorithms assuming that the service offers an interface with the following two functions, which are in practice easily mapped to functions that are supported by the various services that we analyzed: the insertion of an element in a given list *Lst*, denoted by the execution of function `insert`(*Lst*,*ElementID*,*Value*), where *Lst* identifies the list being accessed, *ElementID* denotes the identifier of the element being added (which can be an identifier generated by the centralized service or a unique identifier generated by our Middleware), and *Value* stands for the value of the element being added to the list; and the access to the contents of a list, denoted by the execution of function `get`(*Lst*), where *Lst* identifies the list being read by the client.

When the client accesses a list *Lst* for the first time, a special initialization procedure is triggered internally by our Middleware (Alg. 1), which initializes the local state regarding the accesses to *Lst*. The initialization is straightforward: it creates the object *lstState* that maintains all relevant information to manage the accesses to *Lst* (line 2). This state

Algorithm 1: Initialization of local state

```
1: upon init(Lst) do
2:     lstState ← init()
3:     lstState.insertSet ← {}
4:     lstState.localView ← {}
5:     lstState.lastTimestamp ← 0
6:     lstState.insertCounter ← 0
7:     listStates[Lst] ← lstState
```

Algorithm 2: Read Your Writes

```
1: function insert(Lst, ElementId, Value) do
2:     lstState ← listStates[Lst]
3:     Element e ← init()
4:     e.v ← Value
5:     e.id ← ElementId
6:     e.timestamp ← obtainServiceTimeStamp()
7:     SERVICE.insert(Lst, ElementId, e)
8:     lstState.insertSet ← e ∪ lstState.insertSet

9:  function get(Lst) do
10:    lstState ← listStates[Lst]
11:    sl ← SERVICE.get(Lst)
12:    sl ← orderByTimestamp(sl)
13:    sl ← addMissingElementsToSL(sl, lstState.insertSet, lstState.lastTimestamp)
14:    sl ← purgeOldElementFromSL(sl, lstState.insertSet, lstState.lastTimestamp)
15:    lstState.lastTimestamp ← getLastTimestamp(sl)
16:    return removeMetadata(subList(sl, 0, N))
```

is composed by the sets **insertSet** and **localView** that were discussed previously, and that are initially empty (lines $3-4$). Furthermore, two other variables are initialized, **lastTimestamp**, which is used to maintain information regarding elements that were removed from the previously discussed sets, and **insertCounter**, which tracks the number of inserts performed by the local client in the context of the current session. Both of these variables have an initial value of zero (lines $5-6$). Finally, the *lstState* variable is stored in a local map, associated to the list *Lst* (line 7). Next, we explain how this local state is leveraged by our algorithms to enforce the various session guarantees.

Read Your Writes: The Read Your Writes (RYW) session guarantee requires that, in a session, any read observes all writes previously executed by the same client. More precisely, for every set of insert operations W made by a client c over a list L in a given session, and set S of elements from list L returned by a subsequent get operation of c over L, we say that RYW is violated if and only if $\exists x \in W : x \notin S$.

This definition, however, does not consider the case where only the N most recent elements of a list are returned by a get operation. In this case, some writes of a given client may not be present in the result if more than N other insert operations have been performed (by client c or any other client). Considering that the list must hold the most recent writes, a RYW anomaly happens when a get operation returns an older write performed by the client but misses a more recent one. More formally, given two writes x, y over list L executed in the same client session, where x was executed before y, an anomaly of RYW happens in a get that returns S when $\exists x, y \in W : x \prec y \wedge y \notin S \wedge x \in S$.

Alg. 2 presents our algorithm for providing RYW. To avoid the anomaly described above, the idea is to store, locally at the client, all elements that are inserted by the local

Algorithm 3: Monotonic reads

```
1:  function insert(Lst, ElementID, Value) do
2:      Element e ← init()
3:      e.id ← ElementID
4:      e.v ← Value
5:      SERVICE.insert(Lst, ElementID, e)

6:  function get(Lst) do
7:      lstState ← listStates[Lst]
8:      sl ← SERVICE.get(Lst)
9:      lstState.localView ← appendNewElementsToTop(sl, lstState.localView)
10:     return removeMetadata(subList(lstState.localView, 0, N))
```

client in the list and add them to the result of get operations. In the insert operation, the inserted element is stored locally by the client (line 8). Additionally, our algorithm stores some metadata in the object before performing the insert operation over the centralized service (lines $5-6$). This information represents, respectively, the identifier of the element and a timestamp for the insert operation. The element identifier is used to uniquely identify the writes. The timestamp and element identifier allow for totally ordering all entries in the **insertSet**, with the order being approximately that of the real-time order of execution. Note that the operation in line 12 also checks if the timestamps retrieved from the service in the same session are monotonically increasing, and, if not, enforces that property by overwriting the returned timestamp with an increment of the most recent one; this is important to avoid reordering events from the same session in case the timestamp provided by the server does not increase monotonically for some reason.

For executing a get operation (line 9) our algorithm starts by executing the get operation over the service (line 11). Then, the returned list (*sl*) is ordered (line 12) and all elements of the local **insertSet** that are missing in the list are added to the list, keeping it ordered (line 13). Before returning the most recent N elements (with no metadata) (line 16), our algorithm removes old session elements from the *sl* list and updates the **lastTimestamp** variable with the timestamp of the oldest element of the client session returned to the client (lines $14-16$).

Monotonic Reads: This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. To define this in our scenario where a truncated list of N recent elements is returned, we say that MR is violated when a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists x, y \in S_1 : x \prec y \; in \; S_1 \wedge y \notin S_2 \wedge x \in S_2$, where $x \prec y$ means that element x appears in S_1 before y.

To avoid this anomaly, our algorithm (presented in Alg. 3) resorts to the **localView** variable to maintain information regarding the elements (and their respective order) observed by the client in previous get operations. Therefore, when the client issues a get operation, our Middleware issues the get command over the centralized service (line 8) and then updates the contents of its **localView** with any elements that are returned by the service and that were not yet within the **localView** (line 9). These new elements are appended to the start of the list, as they are assumed to be more recent than those of the current **localView**.

The algorithm terminates by returning to the client the N most recent elements in the **localView**. These elements are exposed to the client without any of the metadata added

Algorithm 4: Monotonic Writes

```
1:  function insert(Lst, ElementID, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.id ← ElementID
5:      e.v ← Value
6:      e.clientSession ← getClientSessionID()
7:      e.sessionCounter ← lstState.insertCounter++
8:      SERVICE.insert(Lst, ElementID, e)

9:  function get(Lst) do
10:     lstState ← listStates[Lst]
11:     sl ← SERVICE.get(Lst)
12:     sl ← sortElementsBySessionCounters(sl)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     return removeMetadata(sl)
```

by our algorithms (line 10). Note that in this case the insert operation only issues the corresponding insert command with additional metadata on the centralized service (lines $1-5$).

A limitation of this algorithm is that it causes the **localView** set to grow indefinitely. To avoid this, we associate with each element inserted in the list a timestamp (obtained from the centralized service). This timestamp allows us to remove from the **localView** any element with a timestamp smaller than the timestamp of the oldest element that was in the last return to the client. We omit this from Alg. 3 for readability.

Monotonic Writes: This session guarantee requires that writes issued by a given client are observed in the order in which they were issued by all clients. More precisely, if W is a sequence of write operations made by client c up to a given instant, and S is a sequence of write operations returned in a read operation by any client, a MW anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W: $\exists x,y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

However, this definition needs to be adapted for the case where only N elements of a list are returned by a get operation. In this case, some session sequences may be incomplete, because older elements of the sequence may be left out of the truncated list of N returned elements. Thus, we consider that older elements are eligible to be dropped from the output, provided that we ensure that there are no gaps in the session subsequences and that the write order is respected, before returning to the client. Formally, we can redefine MW anomalies as follows, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x,y,z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x,y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

Alg. 4 presents the algorithm employed by our Middleware to enforce the MW session guarantee. We avoid the anomaly described above by adding metadata to each insert operation (lines $1-8$) in the form of a unique client session id (*clientSession* – line 6) and a counter (local to each client and session) that grows monotonically (*sessionCounter* – line 7). This information allows us to establish a total order of inserts for each client session.

This metadata is then leveraged during the execution of a get operation (lines $9-14$) in the following way. After reading the current list from the service (line 11), we simply order the elements in the read list (*sl*) to ensure that all elements respect the partial orders for each client session (line 12). Finally, an additional step is required to ensure that

Algorithm 5: Write Follows Read

```
1:  function insert(Lst, ElementID, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.id ← ElementID
5:      e.v ← Value
6:      e.cutTimestamp ← obtainCutTimestamp(lstState.localView)
7:      e.dependencies ← projectElementIdentifiers(lstState.localView)
8:      e.timestamp ← obtainIncreasingServiceTimeStamp(lstState.localView)
9:      SERVICE.insert(Lst, ElementID, e)

10: function get(Lst) do
11:     lstState ← listStates[Lst]
12:     sl ← SERVICE.get(Lst)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     cutTimestamp ← highestCutTimestamp(sl)
15:     sl ← removeElementsBelowCutTimestamp(sl, cutTimestamp)
16:     lstState.localView ← appendNewElementsByTimestamp(ls, lstState.localView)
17:     lstState.localView ← purgeOldElements(lstState.localView)
18:     return removeMetadata(sl)
```

no element is missing in any of these partial orders. To ensure this, whenever a gap is found within the elements of a given client session, we remove all elements whose *sessionCounter* is above the one of any of the missing elements.

The get operation returns the contents that are left in the list *sl* without the metadata added by our algorithms (line 14). Note that in this case we might return to the client a list of elements with a size below N. We could mitigate this behavior by resorting to the contents of the **localView** as we did in the algorithm to enforce MR. However, we decided to provide the minimal behavior to enforce each of the session guarantees in isolation.

Write Follows Read: This session guarantee requires that the effects of a write observed in a read by a given client always precede the writes that the same client subsequently performs. To formalize this definition, and considering that the service only returns at most N elements in a list, if S_1 is a sequence returned by a read invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a violation of the WFR anomaly happens when: $w \in S_2 \wedge \exists x, y \in S_1 : x \prec y \; in \; S_1 \wedge y \notin S_2 \wedge x \in S_2$.

Our algorithm to enforce this session guarantee is depicted in Alg. 5. The key idea to avoid this anomaly is to associate with each insert the direct list of dependencies of that insert, i.e, all elements previously observed by the client performing the insert (line 7). Evidently, this solution is not practical, since this list could easily grow to include all previous inserts performed during the lifetime of the system. To overcome this limitation, we associate with each insert a timestamp based on the clock of the service, but with the restriction of being strictly greater than the timestamp of any of its direct dependencies (line 8). Furthermore, we also associate with each insert a cut timestamp, that defines the timestamp of its last explicit dependency, i.e, the dependencies registered in the dependency list (line 6). The cut timestamp implicitly defines every element with a higher timestamp to be a dependency of that insert operation. By combining these different techniques, we ensure that the explicit dependency list associated with an insert has at most N elements (which is the size of the **localView** maintained by our Middleware).

Since only N elements of a list are returned by a get operation, the older dependencies

may be left out of the sequence that is returned. When this happens, it is safe to consider that these dependencies were dropped from the window that is returned, provided that we ensure that, for each element that is returned, all dependencies that are more recent than the oldest element are also returned.

In the get operation we leverage this metadata to do the following: we start by reading the contents of the list from the service (line 12) and then over this list we remove any insert whose dependencies are missing. Thus, we only remove inserts whose missing dependencies have a timestamp above the insert cut timestamp. We then compute a cut timestamp for the obtained list *sl* (line 13) that is the highest cut timestamp among all elements in *sl*. We use this timestamp to remove from *sl* any element whose creation timestamp falls below the computed cut timestamp. Finally, before returning to the client the elements that remain in *sl* without the additional metadata (line 18) we update and garbage collect old entries from the **localView** (lines $16-17$).

Similarly to the previous algorithm, the service might return a number of elements that is lower than N. In this case, to ensure that we always return N elements, we need to obtain the missing dependencies using a get operation that returns a single element (if supported by the service). In our implementation, we avoided this solution because it is prone to triggering a violation of the API rate limits. Again, an alternative way to address this is by, after reading the list from the service, merging its contents with those in the **localStore** and enforcing an order that is compatible with the timestamp of each element. However, for simplicity, we omitted this from our algorithms.

Combining multiple session guarantees: Considering the algorithms to enforce each of the session guarantees discussed above, we can now summarize how to combine them. In a nutshell, it suffices for our Middleware to, on insert operations, add the metadata used by each of the individual algorithms according to the guarantees configured by the application developer. Correspondingly, upon the execution of a get operation, our Middleware must perform the transformations over the list obtained from the service (*sl*) prescribed by each of the individual algorithms. Furthermore, all metadata added to each element must also be removed before exposing data to the client application.

Final Remarks This work shows that it is possible to enforce different consistency properties, in particular session guarantees for third party applications that access online services through their public APIs. We do so without explicit support from the service architecture, and without assuming that the service itself provides any of these guarantees. Our solution relies on a thin Middleware layer that executes on the client side, and intercepts all interactions of the client with the online service. We have presented different algorithms to enforce each of the well known session guarantees. Furthermore, our algorithms follow a simple structure that allows to combine then easily.

We have developed a prototype that we used to evaluate our approach, showing that we can enforce session guarantees with a modest overhead both in terms of user-perceived latency and communication with the centralized service. Complete results are presented in [28] (attached in Appendix).

2.4 Relation to use cases

(a) Monitoring Guifi.net community network

UPC's scenario of interest in the *heavy edge* domain consists on monitoring the Guifi.net Community Network, acquiring data from its network devices and distributing them in a replicated storage for further processing (validation, aggregation, averaging, etc.).

The UC2 application from WP2 consists of a large number of monitoring devices (between a few tens and a hundred), geographically distributed all over the Guifi.net network (in particular, at the network edges), backed by more powerful devices placed in local data centers. These devices periodically probe a given list of network nodes and retrieve a set of data from them, either by direct observation (i.e. performing ping tests to find the round-trip time or estimate up-time) or as reported by the network devices (e.g. network traffic through an interface). For any given network node, more than one monitor may be performing observations at a given time. This redundancy adds fault tolerance and resilience to the monitoring application (against network partitions, failing monitors, etc.) but may come at the expense of adding concurrency issues to the application, like two monitors watching a given node and reporting conflicting or incoherent data.

A multi-probe, distributed and coordinated network monitoring system can provide much more information than several independent, single-probe ones. For example, a nodes state flipping on-line/off-line can be an indicator of a hardware failure (e.g. a defective network interface). However, by observing the phenomenon from different locations, more precise or realistic information can be obtained (for instance, if the node's state flips are only observed at some of the monitors, the phenomenon can be instead a symptom of a network failure or misconfiguration somewhere else). Smart aggregation of the whole monitoring data, or parts of it, therefore, can help identifying a wider variety of network issues or at least locating them with increased precision. However, this operation must be performed in an efficient way, both in terms of computational resources and data storage requirements.

Beyond the nodes' on/off-line states and availability, it is of great interest retrieving additional data provided by network nodes themselves, such as network traffic, system load, wireless links quality (for those network devices with radio interfaces), etc. These data are important for both monitoring, to help on the diverse network operation and maintenance tasks required, and billing purposes, to charge or compensate each of the network participants proportionally to the resources they contribute and the ones they use. Given the economic interest involved in this operation, it is important to ensure the availability, correctness and completeness of these data, therefore requiring the distribution and replication of the probes and their cooperation. In this sense, network measurements are to be performed periodically, once per minute by each monitor. It is expected that each of the measurements available at the different monitors will very similar, but slightly different to the others (because of the fact that performing a measurement incurrs in generating traffic through the network, for instance). Therefore, in order to reach a unique and valid measurement on which to base later decision making, a mechanism to merge all the measurements is required.

The development of the network monitoring application envisioned heavily depends on the runtime system and the programming model for edge computing developed in WP3 and WP4, respectively. In particular, the application depends on an infrastructure enabled with distributed replicas and persistent data storage that can guarantee the

correctness of data, even when concurrent write operations take place. These replicas, additionally, must adapt to the heterogeneity of the nodes they are hosted on, both in computational and storage resources terms, as in network resources available due to their different placement. The application also depends on the ability to perform computations with these data (aggregating, averaging, merging, etc.) without having to care about its local availability or readiness, but delegating this task to the underlying runtime system. Furthermore, because of the always growing nature of the collected dataset and taking into account that as time passes, older data become less interesting than the newly captured ones, data summarizing algorithms are required to ensure the storage capacity stays within acceptable limits, while ensuring that coherence and meaningful information are preserved.

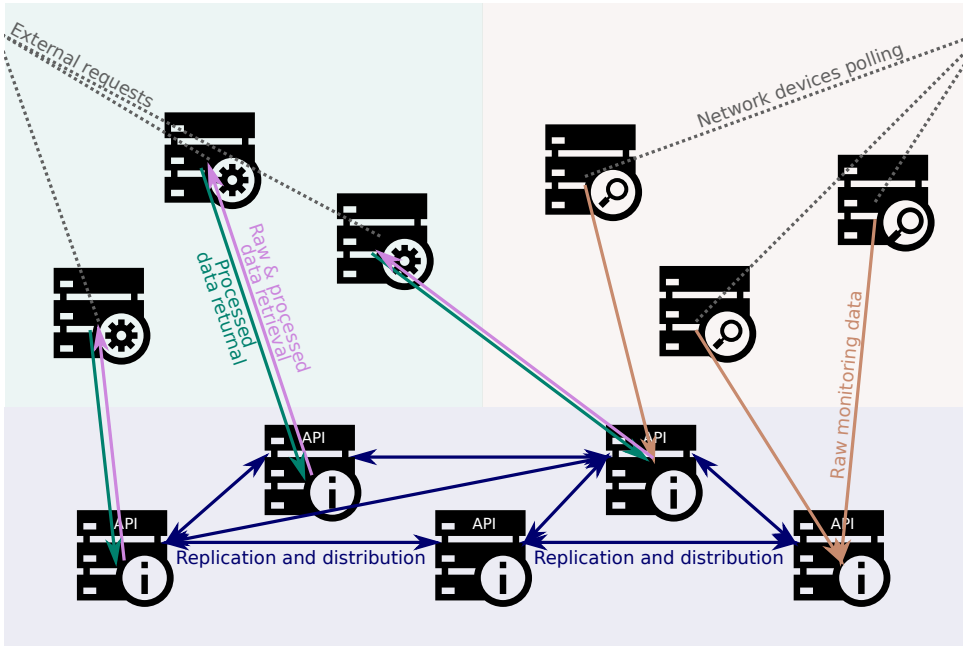


Figure 2.22: Envisioned components of the network monitoring data storage and replication application.

From the software and applications development point of view, a RESTful API is the preferred mechanism to interact with the underlying infrastructure providing all the abovementioned features and requirements. This adds an abstraction layer between the infrastructure and the application that allows developing and operating them independently, or even using two different implementations (e.g. different programming languages, different versions or shipped features).

Figure 2.22 shows a high-level vision of the application development. On the right side, on the orange area, are the network monitoring instances that collect diverse types of data from the network devices. These data are sent as they are to the *heavy edge* infrastructure, that manages the distributed and replicated storage transparently to the application. This infrastructure, actually, is not an intrinsic part of the application, but an *enabler*. All the communication with the infrastructure is performed through the API it provides. On the left, on green, there are generic services or applications that leverage the infrastructure to retrieve the raw data sent by the monitors in order to process it and that, when done, may return the processed information back to the infrastructure.

These generic services or applications may include graphs and stats generation, network management alerts, billing and accounting, etc.

(b) Building a weakly-consistent datastore index

Cloud storage systems are able to scale to very large amounts of data and provide high throughput and low latency reads and writes. To achieve this, storage systems typically expose a simple GET/PUT API that allows access to data only through their primary key. However, applications often require the ability to retrieve stored objects by performing queries on attributes other than their primary key. A common approach to address this is to maintain indexes on these secondary attributes. Geo-distributed, weakly-consistent storage systems pose varying requirements to the design of secondary indexing systems. These requirements include supporting efficient search on large volumes of data, enabling multiple ingest points for updates and queries on different replicas, and incurring low overhead to the storage system's performance. Additionally, different use cases have different requirements and need to optimise different metrics such as storage overhead of indexes or query response time. It is challenging for a secondary indexing system to be optimal on all dimensions. Indeed, existing approaches [24, 35, 62] are designed with specific workloads and storage system characteristics in mind.

Modular secondary indexing [66] is an ongoing work on an approach for expanding geo-distributed cloud storage systems to support secondary attribute indexing and search. The key insight is that a modular system architecture can enable indexing systems to be flexible and adjustable. Different system configurations can make different trade-offs be optimised for different requirements.

A modular indexing system is assembled from indexing and query processing modules, called Query Processing Units (QPUs). QPUs operate as services that receive and process queries. Individual QPUs perform basic indexing and query processing tasks such as maintaining indexes, caching search results, and federating search over a geo-distributed system. Different QPU classes exist based on their functionality, while all exposing a common interface. Indexing systems are built by interconnecting QPUs in a network, using their common interface. Search queries are processed by being routed through the QPU network; individual QPUs partially process a given query, decompose it to sub-queries, forward them through the network, and combine the retrieved results. Different QPU network configurations can be constructed by the same building blocks, each optimised for different use cases and requirements.

Advantages An advantage of this approach is that it can enable indexing systems to be dynamically constructed. QPU networks can be initialised with a simple configuration and then dynamically adapted to the update and query workloads. Moreover, modular indexing systems can be used to dynamically adjust the amount computation resources available for indexing and query processing. Query processing units operate as services and are not bound to physical machines. This enables systems to collocate multiple under-utilised QPUs in the same physical machines, or migrate highly loaded QPUs to new available machines. These mechanisms can be used in order to dynamically adjust indexing according to query and update workloads, according to specific application requirements.

Modular indexing at the edge Modular secondary indexing could be used for performing indexing and query processing tasks at the edge. User data stored in cloud storage systems could be cached in replicas at the edge (user devices), in order to improve latency and allow temporary offline operations. In such systems, index computation could be performed at edge, using user computational resources. User machines would locally maintain inverted indexes of data stored in their cache. These per-client partial indexes would be asynchronously propagated to other replicas and merged to a global index. This can be achieved by placing some parts of the QPU network at user devices, such as QPUs that maintain local inverted indexes, and other parts at the data center. Indexing at the edge can reduce computation load at the core of the system, improve availability in case of partitioning, and enable sophisticated indexing techniques. Moreover, modular indexing could be used in light edge scenarios, such as query processing in sensor networks. In this case, maintaining indexes is challenging due to storage and power constraints, and an acquisitional query processing approach can be applied. Our approach can be used in these cases, as QPUs can process queries by directly scanning or acquiring query results from the underlying data store (or sensor in this case).

Bounding search result staleness In the described system, partial indexes constructed at the edge would be propagated to other replicas asynchronously. As a consequence, indexes could temporarily diverge from the state of the data store and result to stale search results. To address this, we expand our modular indexing framework to enable users to bound the staleness of search results. This can be achieved using two mechanisms. First, indexing systems can monitor the amount of divergence between the indexes and the data store, and dynamically make more computational resources available for index maintenance. A second mechanism can allow users to specify staleness bounds to individual queries. The system can then synchronously pull data from other replicas and perform additional computations in order to reach the specified bound, before responding to a query. Since acquiring less stale search results requires additional computations, applications can use this mechanism to make a trade-off between query response time and query result freshness.

Implementation This work is at an early stage. As a next stage of our work, we plan to implement and evaluate this approach for the described edge use case. We plan to use AntidoteDB as a reference platform for our implementation. Inter-dc replication and CRDT support are features that make AntidoteDB useful for our modular indexing implementation. AntidoteDB's inter-dc communication mechanism, which is used for inter-dc replication, can be used for communication among QPUs. Secondary indexes can be implemented using CRDTs in order to be able to concurrently ingest updates from multiple replicas without coordination. Furthermore, transactional causal+ consistency can be used for causally updating multiple secondary indexes when a data object is updated.

(c) A file system on AntidoteDB

Some of today's most used distributed applications are built atop the long-lived and well-known abstraction of POSIX file systems. Thus, it is paramount to make sure that this distributed version of file systems provide reasonable performance while respecting, as much as possible, the semantics of their non-distributed counterparts. With AntidoteFS

we take a step in that direction: we leverage the research on CRDTs by layering our file system on top of AntidoteDB. AntidoteDB offers transactional causal consistency and a rich toolbox of ready-made CRDTs. The challenge is then twofold: 1. emulating a traditional, POSIX data model with CRDTs, and 2. respecting the file system invariants by using the least possible amount of coordination. In the following, we briefly describe each of these challenges, and summarize the solution we adopted for our reference implementation.[9]

A POSIX data model with CRDTs A first, naive approach to design a file system data model using CRDTs would consist in mirroring its file and folder hierarchical structure using CRDT Maps and Registers. Unfortunately, this approach falls short of being flexible enough to support data-heavy operations, and it would require full support for nested CRDTs in AntidoteDB. An improvement over this approach consists in having a CRDT Map acting as index of all possible file system paths, and pointing to other Maps storing the data of individual files or folder. This design presents a scalability bottleneck in the path map. The third approach we identified takes inspiration from the actual inode data structure in POSIX-compliant file systems. We model each inode as a CRDT Map containing the exact information of POSIX inodes, about ownership, permissions, and the mutual linking of files to their containing folders. Figure 2.23 shows a first instance of this design, which we implemented in the AntidoteFS prototype.

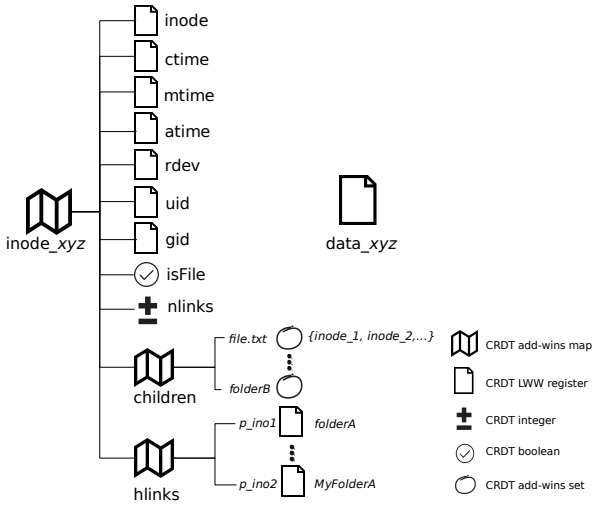


Figure 2.23: Modeling with CRDTs the inode data structure of POSIX file systems.

Respecting POSIX invariants with minimum coordination Executing concurrent file system operations in distributed settings can lead to anomalies in the overall structure of the file system. Namely, an erroneous implementation would allow breaking the tree-like structure of the file system, by engendering cycles or invalid references. Recent research suggests that only a subset of file system user operations need coordination in distributed settings to preserve the POSIX file system invariants. This result comes from applying the CEC analysis (see Sec. 2.2(b)) to a formal specification of the file system

[9] https://github.com/SyncFree/antidote-fs

structure and its invariants. The main practical outcome for our implementation is that we will have to enforce two different consistency semantics for the file system operations: causal consistency for operations that don't require coordination, and a stronger consistency semantics for operations that require further coordination to respect the file system invariants. This is object of ongoing work from the theoretical standpoint — to formalize the overall semantics of the file system – and from the practical standpoint — to implement this dual replication protocol within AntidoteFS. This is, essentially, a direct application of the *Just-Right Consistency* approach outlined in D4.1.

3 Papers and publications

The following is the list of peer-reviewed publications where the work towards the Deliverable has been presented.

- Gonçalo Cabrita and Nuno M. Preguiça. Non-uniform replication. In *Proceedings of OPODIS 2017*, 2017.
- Filipe Freitas, João Leitão, Nuno Preguiça, and Rodrigo Rodrigues Gonçalo Cabrita and Nuno M. Preguiça. Fine-Grained Consistency Upgrades for Online Services. In *Proceedings of Symposium on Reliable Distributed Systems (SRDS'2017)*, 2017.
- Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Blotter: Low latency transactions for geo-replicated storage. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, pages 263–272, 2017.
- Zhongmiao Li, Peter Van Roy, and Paolo Romano. Exploiting speculation in partially replicated transactional data stores. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24 - 27, 2017*, page 640, 2017.
- Bernardo Ferreira, João Leitão, and Henrique Domingos. Multimodal indexable encryption for mobile cloud-based applications. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 213–224, 2017.
- Peter Zeller. Testing properties of weakly consistent programs with repliss. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia, April 23 - 26, 2017*, pages 3:1–3:5, 2017.
- Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Devaki Akkoorath, Annette Bieniusa, João Leitão, and Nuno M. Preguiça. Fmke: a real-world benchmark for key-value data stores. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia, April 23 - 26, 2017*, pages 7:1–7:4, 2017.
- Gonçalo Marcelino, Valter Balegas, and Carla Ferreira. Bringing hybrid consistency closer to programmers. Im *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia, April 23 - 26, 2017*, pages 6:1–6:4, 2017.

4 Software

The following software artifacts are publicly available:

- AntidoteDB website with documentation http://antidotedb.com
- AntidoteDB code https://github.com/SyncFree/antidote
- Antidote Query Language (AQL) https://github.com/JPDSousa/AQL
- Speculative execution https://github.com/marsleezm/STR
- Repliss tool https://softech.cs.uni-kl.de/repliss/
- CEC tool https://github.com/LightKone/CEC
- File system on AntidoteDB https://github.com/SyncFree/antidote-fs

Access to the code of Repliss and CEC can be obtained by the project coordinator.

References

[1] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995.

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *HOTOS*, pages 13–13, Berkeley, CA, USA, 2015. USENIX Association.

[3] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*, pages 405–414. IEEE Computer Society, 2016.

[4] ANSI. X3. 135-1992, American National Standard for Information Systems-Database Language-SQL, 1992.

[5] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[6] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD*, pages 27–38, New York, NY, USA, 2014. ACM.

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

[8] 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.

[9] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiçca. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *Proc. of the Symposium on Reliable Distributed Systems (SRDS'15)*, Set 2015.

[10] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *NSDI*, volume 6, pages 5–5, 2006.

[12] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. *CoRR*, abs/cs/0701157, 2007.

[13] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control; Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983.

[14] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, San Jose, CA, 2013. USENIX.

[16] Gonçalo Cabrita and Nuno Preguiça. Non-uniform replication, 2017.

[17] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[18] James C. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[19] Couchbase, December 2015.

[20] Tyler Crain and Marc Shapiro. Designing a causally consistent protocol for geo-distributed partial replication. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 6:1–6:4, New York, NY, USA, 2015. ACM.

[21] 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.

[22] Phil Dixon. Shopzilla site redesign: We get what we measure. In *Velocity Conference Talk*, 2009.

[23] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*, pages 173–184. IEEE Computer Society, 2013.

[24] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[25] David Ferraiolo and Richard Kuhn. Role-Based Access Control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[26] Bernardo Ferreira, João Leitão, and Henrique Domingos. Multimodal Indexable Encryption for Mobile Cloud-based Applications. In *Proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks. DSN'17*. IEEE, 2017.

[27] F. Freitas, J. Leit ao, N. Preguiça, and R. Rodrigues. Characterizing the consistency of online services (practical experience report). In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 638–645, June 2016.

[28] F. Freitas, J. Leitão, N. Preguiça, and R. Rodrigues. Fine-Grained Consistency Upgrades for Online Services. In *Proceedings of the 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 1–10, Sept 2017.

[29] Gonçalo Cabrita. AntidoteDB NuCRDT Module.

[30] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[31] Goetz Graefe et al. Controlled lock violation. In *SIGMOD*. ACM, 2013.

[32] Xin Jin, Ram Krishnan, and Ravi Sandhu. *A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[33] Xin Jin, Ravi Sandhu, and Ram Krishnan. *RABAC: Role-Centric Attribute-Based Access Control*, pages 84–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[34] Evan Jones et al. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*. ACM, 2010.

[35] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 57–70, Denver, CO, 2016. USENIX Association.

[36] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, 2013.

[37] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[38] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

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

[40] Greg Linden. Make data useful, 2006.

[41] 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.

[42] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, 2013.

[43] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[44] Gonçalo Marcelino, Valter Balegas, and Carla Ferreira. Bringing hybrid consistency closer to programmers. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, pages 6:1–6:4, New York, NY, USA, 2017. ACM.

[45] MongoDB for GIANT Ideas – MongoDB, December 2015.

[46] Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 263–272, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[47] Sreeja S Nair. Evaluation of the CEC (Correct Eventual Consistency) Tool. Research Report RR-9111, Inria Paris ; LIP6 UMR 7606, UPMC Sorbonne Universités, France, November 2017.

[48] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[49] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís E. T. Rodrigues. GMU: genuine multiversion update-serializable partial data replication. *IEEE Trans. Parallel Distrib. Syst.*, 27(10):2911–2925, 2016.

[50] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. On Brewing Fresh Espresso: Linkedin's Distributed Data Serving Platform. In *SIGMOD*, pages 1135–1146, New York, NY, USA, 2013. ACM.

[51] Riak KV, December 2015.

[52] Paolo Romano et al. On speculative replication of transactional systems. *J. Comput. Syst. Sci.*, 80(1), February 2014.

[53] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Proc. of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS 2013)*, pages 163–172, 2013.

[54] Pierangela Samarati and Sabrina Capitani de Vimercati. *Access Control: Policies, Models, and Mechanisms*, pages 137–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[55] Gregory Saunders, Michael Hitchens, and Vijay Varadharajan. *An Analysis of Access Control Models*, pages 281–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[56] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *2010 29th IEEE Symposium on Reliable Distributed Systems*, pages 214–224, Oct 2010.

[57] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[58] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[59] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. Consistency in 3d. In Josée Desharnais and Radha Jagadeesan, editors, *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, volume 59 of *LIPIcs*, pages 3:1–3:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[60] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.

[61] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011.

[62] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. Replex: A scalable, highly available multi-index data store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 337–350, Berkeley, CA, USA, 2016. USENIX Association.

[63] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[64] Alejandro Zlatko Tomsic, Manuel Bravo, and Marc Shapiro. Distributed transactional reads: the strong, the quick, the fresh & the impossible, 2018.

[65] Twitter, inc.

[66] D. Vasilas. Search on Secondary Attributes in Geo-Distributed Systems. *ArXiv e-prints*, January 2018.

[67] Mathias Weber and Annette Bieniusa. ACGreGate: A framework for practical access control for applications using weakly consistent databases. *CoRR*, abs/1704.05320, 2018.

[68] Marek Zawirski, Nuno M. Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In Rodger Lea, Sathish Gopalakrishnan, Eli Tilevich, Amy L. Murphy, and Michael Blackstock, editors, *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, pages 75–87. ACM, 2015.

[69] Peter Zeller. Testing properties of weakly consistent programs with repliss. In Annette Bieniusa and Alexey Gotsman, editors, *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2017, Belgrade, Serbia, April 23 - 26, 2017*, pages 3:1–3:5. ACM, 2017.

[70] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP, pages 276–291, 2013.

A Publications

Blotter: Low Latency Transactions for Geo-Replicated Storage

Henrique Moniz[1*] João Leitão[2] Ricardo J. Dias[3] Johannes Gehrke[4]
Nuno Preguiça[2] Rodrigo Rodrigues[5]

[1]Google [2]NOVA LINCS & FCT, Universidade NOVA de Lisboa [3]NOVA LINCS & SUSE Linux GmbH
[4]Microsoft [5]INESC-ID & Instituto Superior Técnico, Universidade de Lisboa

ABSTRACT

Most geo-replicated storage systems use weak consistency to avoid the performance penalty of coordinating replicas in different data centers. This departure from strong semantics poses problems to application programmers, who need to address the anomalies enabled by weak consistency. In this paper we use a recently proposed isolation level, called Non-Monotonic Snapshot Isolation, to achieve ACID transactions with low latency. To this end, we present Blotter, a geo-replicated system that leverages these semantics in the design of a new concurrency control protocol that leaves a small amount of local state during reads to make commits more efficient, which is combined with a configuration of Paxos that is tailored for good performance in wide area settings. Read operations always run on the local data center, and update transactions complete in a small number of message steps to a subset of the replicas. We implemented Blotter as an extension to Cassandra. Our experimental evaluation shows that Blotter has a small overhead at the data center scale, and performs better across data centers when compared with our implementations of the core Spanner protocol and of Snapshot Isolation on the same codebase.

Keywords

Geo-replication; non-monotonic snapshot isolation; concurrency control.

1. INTRODUCTION

Many Internet services are backed by geo-replicated storage systems, in order to keep data close to the end user. This decision is supported by studies showing the negative impact of latency on user engagement and, by extension, revenue [15]. While many of these systems rely on weak consistency for better performance and availability [10], there is also a class of applications that require support for strong consistency and transactions. For instance, many applications within Google are operating on top of Megastore [3], a system that provides ACID semantics within the same shard, instead of Bigtable [7], which provides better performance but weaker semantics. This trend also motivated the development of Spanner, which provides general serializable transactions [9], and sparked other recent efforts in the area of strongly consistent geo-replication [28, 24, 22, 30, 16, 23].

In this paper, we investigate whether it is possible to further cut the latency penalty for ACID transactions in a geo-replicated systems, by leveraging a recent isolation proposal called Non-Monotonic Snapshot Isolation (NMSI) [24]. We present the design and implementation of Blotter, a transactional geo-replicated storage system that achieves: (1) at most one round-trip across data centers (assuming a fault-free run and that clients are proxies in the same data center as one of the replicas), and (2) read operations that are always served by the local data center. Additionally, when the client is either co-located with the Paxos leader or when that leader is in the closest data center to the client, Blotter can operate in a single round-trip to the *closest* data center.

To achieve these goals, Blotter combines a novel *concurrency control* algorithm that executes at the data center level, with a carefully configured Paxos-based replicated state machine that replicates the execution of the concurrency control algorithm across data centers. Both of these components exploit several characteristics of NMSI to reduce the amount of coordination between replicas. In particular, the concurrency control algorithm leverages the fact that NMSI does not require a total order on the start and commit times of transactions. Such an ordering would require either synchronized clocks, which are difficult to implement, even using expensive hardware [9], or synchronization between replicas that do not hold the objects accessed by a transaction [25], which hinders scalability. In addition, NMSI allows us to use separate (concurrent) Paxos-based state machines for different objects, on which we geo-replicate the *commit* operation of the concurrency control protocol.

Compared to a previously proposed NMSI system (Jessy [24]), instead of assuming partial replication we target full replication, which is a common deployment scenario [3, 27, 5]. Our layering of Paxos on top of a concurrency control algorithm is akin to the Replicated Commit system, which layers Paxos on top of Two-Phase Locking [23]. However, by leveraging NMSI, we execute reads exclusively locally, and run parallel instances of Paxos for different objects, instead of having a single instance per shard.

We implemented Blotter as an extension to Cassandra [17]. Our evaluation shows that, despite adding a small overhead in a single data center, Blotter performs much better than Jessy and the protocols used by Spanner, and outperforms in many metrics a replication protocol that ensures SI [12]. This shows that Blotter can be a valid choice when several replicas are separated by high latency links, performance is critical, and the semantic differences between NMSI and SI are tolerated by the application.

*Work done while the author was at NOVA LINCS.

©2017 International World Wide Web Conference Committee (IW3C2), published under Creative Commons CC BY 4.0 License.

ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052603

2. SYSTEM MODEL

Blotter is designed to run on top of any distributed storage system with nodes spread across one or multiple data centers. We assume that each data object is replicated at all data centers. Within each data center, data objects are replicated and partitioned across several nodes. We make no restrictions on how this intra-data center replication and partitioning takes place. We assume that nodes may fail by crashing and recover from such faults. When a node crashes, it loses its volatile state but all data that was written to stable storage is accessible after recovery. We use an asynchronous system model, i.e., we do not assume any known bounds on computation and communication delays. We do not prescribe a fixed bound on the number of faulty nodes within each data center. As we will see, our modular design allows for plugging in different replication protocols that run within each data center. As such, the bounds on faulty nodes depend on the intra-data center replication protocol.

3. NON-MONOTONIC SI

This section specifies our target isolation level, NMSI, and discusses the advantages and drawbacks of this choice. The reason for formalizing NMSI is twofold. First, our specification is simpler than the previous definition [24], thus improving in clarity and readability. Second, some of our key design choices follow naturally from this specification.

3.1 Snapshot isolation revisited

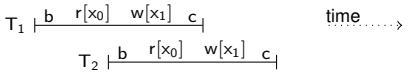
NMSI is an evolution of Snapshot Isolation (SI). Under SI, a transaction (logically) executes in a database snapshot taken at the transaction begin time, reflecting the writes of all transactions that committed before that instant. Reads and writes execute against this snapshot, and, at commit time, a transaction can commit if there are no write-write conflicts with concurrent transactions. (In this context, two transactions are concurrent if the intervals between their begin and commit times overlap.)

To define SI more precisely, we state that for any execution of a system implementing SI, we must be able to create a partial order among the transactions that were executed that (1) explains the values observed by all transactions, with reads returning the value written by the latest transaction, according to this partial order, that wrote to that object; (2) totally orders transactions that write to the same object; and (3) ensures that transactions see a snapshot that reflects all operations that committed before the transaction started. More precisely:

DEFINITION 3.1 (SNAPSHOT ISOLATION (SI)). *An implementation of a transactional system obeys SI if, for any trace of an execution of that system, there exists a partial order $\prec$ among transactions that obeys the following rules, for any pair of transactions t_i and t_j in that trace:*

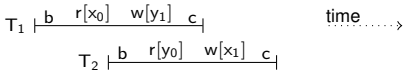
1. *if t_j reads a value for object x written by t_i then $t_i \prec t_j \wedge \nexists t_k$ writing to $x : t_i \prec t_k \prec t_j$*
2. *if t_i and t_j write to the same object x then either $t_i \prec t_j$ or $t_j \prec t_i$.*
3. *$t_i \prec t_j$ if and only if t_i commits before t_j begins.*

This definition captures the anomalies that are used in most definitions of SI. In particular, it prevents concurrent transactions from writing to the same object. For example, consider the following non-SI execution[1]:



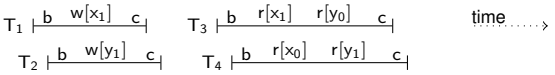
In the above example, transactions T_1 and T_2 write to the same object x and both commit. Such execution is impossible under SI, since two transactions that write to the same object must be ordered according to point number 2 of Definition 3.1, and, for any ordered pair of transactions, the first one must have committed before the start of the second transaction, according to point number 3.

The write-skew anomaly is also captured by Definition 3.1, since concurrent transactions with disjoint write-sets are not ordered. For example, the following execution meets SI, but is not serializable.



3.2 Specification of NMSI

NMSI weakens the SI specification in two ways. First, the snapshots against which transactions execute do not have to reflect the writes of a monotonically growing set of transactions. In other words, it is possible to observe what is called a "long fork" anomaly, where there can exist two concurrent transactions t_a and t_b that commit, writing to different objects, and two other transactions that start subsequently, where one sees the effects of t_a but not t_b, and the other sees the effects of t_b but not t_a. The next figure exemplifies an execution that is admissible under NMSI but not under SI, since under SI both T_3 and T_4 would see the effects of both T_1 and T_2 because they started after the commit of T_1 and T_2.



This relaxation affects Definition 3.1 by turning the equivalence in point 3 into an implication, i.e., it becomes:

3' if $t_i \prec t_j$ then t_i commits before t_j begins.

Second, instead of forcing the snapshot to reflect a subset of the transactions that committed at the transaction begin time, NMSI gives the implementation the flexibility to reflect a more convenient set of transactions in the snapshot, possibly including transactions that committed *after* the transaction began. This property, also enabled by serializability, is called *forward freshness* [24].

Going back to Definition 3.1, we can completely remove point 3, since it is now possible that the snapshot that t sees reflects writes from transactions that commit after t started, as long as the resulting snapshot is valid at some moment before the commit of t, i.e.:

DEFINITION 3.2 (NON-MON. SNAPSHOT ISOL. (NMSI)). *An implementation of a transactional system obeys NMSI if, for any trace of the system execution, there exists a partial order $\prec$ among transactions that obeys the following rules, for any pair of transactions t_i and t_j in the trace:*

1. *if t_j reads a value for object x written by t_i then $t_i \prec t_j \wedge \nexists t_k$ writing to $x : t_i \prec t_k \prec t_j$*
2. *if t_i and t_j write to the same object x then either $t_i \prec t_j$ or $t_j \prec t_i$.*

The example in Figure 1 obeys NMSI but not SI, as the depicted partial order meets Definition 3.2, but it is not possible to create a partial order obeying all three requirements of Definition 3.1.

[1]The notation b, r[x_j], w[y_l], c and a refers to the following operations of a transaction: begin; read version j of object x; write version l of object y; commit; and abort.

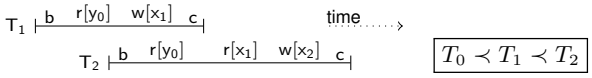


Figure 1: Example execution obeying NMSI but not SI.

3.3 What is enabled by NMSI?

NMSI weakens the specification of SI through the two properties we mentioned previously, which are individually leveraged by the design of Blotter.

The possibility of having "long forks" allows, in a replicated setting, for a single (local) replica to make a decision concerning what data the snapshot should read. This is because, in any highly available design for the commit protocol, there is necessarily the possibility of some replicas not seeing a subset of the most recent commits (since otherwise it would be impossible to provide availability when a data center is unreachable). As such, in a situation where snapshots are based on local information, and a replica in data center $DC1$ sees the writes of t_1 but not t_2, and conversely a replica in $DC2$ sees the writes of t_2 but not t_1, then the two local snapshots taken at each of these replicas can lead to the "long fork" anomaly we mentioned, where two transactions proceed independently. Avoiding this situation would require a serialization between all transaction begin and commit operations.

In the case of "forward freshness", this allows for a transaction to read (in most cases) the most recent version of a given replica, without having to worry about the instant when the transaction began. This not only avoids the bookkeeping associated with keeping track of transaction start times, but also avoids a conflict with transactions that might have committed after the transaction began.

3.4 Discussion: Limitations of NMSI

We analyze in turn the impact of "forward freshness" and "long forks" on programmability. Forward freshness allows a transaction x to observe the effects of another transaction y that committed after x began (in real-time). In this case, the programmer must decide whether this is a violation of the intended application semantics, analogously to deciding whether serializability or strict serializability is the most adequate isolation level for a given application. Long forks allow two transactions to be executed against different branches of a forked database state, provided there are no write-write conflicts. In practice, the main implication of this fact is that the updates made by users may not become instantly visible across all replicas. For example, this could cause two users of a social network to each think that they were the first to post a new promotion on their own wall, since they do not see each other's posts immediately [28]. Again, the programmer must reason whether this is admissible. In this case, a mitigating factor is that this anomaly does not cause the consistency of the database to break. (This is in contrast with the "write skew" anomaly, which is present in both SI and NMSI.) Furthermore, in the particular case of our implementation of NMSI, the occurrence of anomalies is very rare: for a "long fork" to occur, two transactions must commit in two different data centers, form a quorum with a third data center, and both complete before hearing from the other.

Finally, NMSI allows consecutive transactions from the same client to observe a state that reflects a set of transactions that does not grow monotonically (when consecutive transactions switch between two different branches of a long fork). However, in our algorithms this is an unlikely occurrence, since it requires that a client connects through different data centers in a very short time span.

4. ARCHITECTURE OF BLOTTER

The client library of Blotter exposes an API with the expected operations: `begin` a new transaction, `read` an object given its identifier, `write` an object given its identifier and new value, and `commit` a transaction, which either returns commit or abort.

The set of protocols that comprise Blotter are organized into three different components. This not only leads to a modular design, but also allows us to more clearly define the requirements and design choices of each component.

Blotter intra-data center replication. At the lowest level, we run an intra-data center replication protocol, to mask the unreliability of individual machines within each data center. This level must provide the protocols above it with the vision of a single logical copy (per data center) of each data object and associated metadata, which remains available despite individual node crashes. We do not prescribe a specific protocol for this layer, since any of the existing protocols that meet this specification can be used.

Blotter Concurrency Control. (Section 5.) These are the protocols that ensure transaction atomicity and NMSI isolation in a single data center, and at the same time are extensible to multiple data centers by serializing a single protocol step.

Paxos. (Section 6.) This completes the protocol stack by replicating a subset of the steps of the concurrency control protocol across data centers. It implements state machine replication [26, 18] using a careful parameterization of Paxos [19]. However, state machine replication must be judiciously applied to the concurrency control protocol, to avoid unnecessary coordination across data centers.

5. SINGLE DATA CENTER PROTOCOL

5.1 Overview

We start by explaining how we derive the concurrency control protocol from the NMSI requirements.

Partial order $\prec$. We use a multi-version protocol, i.e., the system maintains a list of versions for each object. This list is indexed by an integer version number, which is incremented every time a new version of the object is created (e.g., for a given object x, overwriting x_0 creates version x_1, and so on). In a multi-versioned storage, the $\prec$ relation can be defined by the version number that transactions access, namely if t_i writes x_m and t_j writes x_n, then $t_i \prec t_j \Leftrightarrow m < n$; and if t_i writes x_m and t_j reads x_n, then $t_i \prec t_j \Leftrightarrow m \leq n$.

NMSI rule number 1. Rule number 1 of the definition of NMSI says that, for object x, transaction t must read the value written by the "latest" transaction that updated x (according to $\prec$). To illustrate this, consider the example run in Figure 2. When a transaction $T1$ issues its first read operation, it can read the most recently committed version of the object, say x_i written by $T0$ (leading to $T0 \prec T$). If, subsequently, some other transaction $T2$ writes x_{i+1} ($T0 \prec T2$), then the protocol must prevent $T1$ from either reading or overwriting the values written by $T2$. Otherwise, we would have $T0 \prec T2 \prec T1$, and $T1$ should have read the value for object x written by $T2$ (i.e., x_{i+1}) instead of that written by $T0$ (i.e., x_i). Next, we detail how this is achieved first for reads, then writes, and then how to enforce the rule transitively.

Reading the latest preceding version. The key to enforcing this requirement is to maintain state associated with each object, stating the version a running transaction must read, in case such a restriction exists. In the previous example, if $T2$ writes x_{i+1}, this state records that $T1$ must read x_i.

To achieve this, our algorithm maintains a per-object dictionary data structure (x.*snapshot*), mapping the identifier of a transaction t

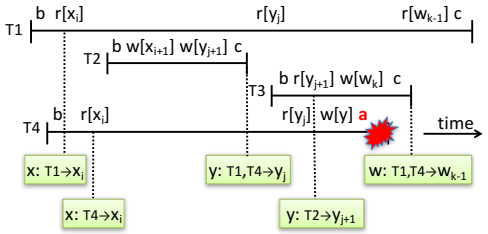


Figure 2: Example run.

to a particular version of x that t either must read or has read from. Figure 2 depicts the changes to this data structure in the shaded boxes at the bottom of the figure. When t issues a read for x, if the dictionary has no information for t, the most recent version is read and this information is stored in the dictionary. Otherwise, the specified version is returned.

In the previous example, $T1$ must record the version it read in the *x.snapshot* variable. Subsequently, when the commit of $T2$ overwrites that version of x, we are establishing that $T2 \not\prec T1$. As such, if $T2$ writes to another object y, creating y_{j+1}, then it must also force $T1$ to read the preceding version y_j. To do this, when transaction $T2$ commits, for every transaction t that read (or must read) an older version of object x (i.e., the transactions with entries in the dictionary of x), the protocol will store in the dictionary of every other object y written by $T2$ that t must read the previous version of y (unless an even older version is already prescribed). In this particular example, $y.snapshot$ would record that $T1$ and $T4$ must read version y_j, since, at commit time, $x.snapshot$ indicates that these transactions read x_i.

Preventing illegal overwrites. In the previous example, we must also guarantee that $T1$ does not overwrite any value written by $T2$. To enforce this, it suffices to verify, at the time of the commit of transaction t, for every object written by t, if T should read its most recent version. If this is the case, then the transaction can commit, since no version will be incorrectly overwritten; otherwise, it must abort. In the example, $T4$ aborts, since $y.snapshot$ records that $T4$ must read y_j and a more recent version exists (y_{j+1}). Allowing $T4$ to commit and overwrite y_{j+1} would lead to $T2 \prec T4$. This breaks rule number 1 of NMSI, since it would have required $T1$ to read x_{i+1} written by $T2$, which did not occur.

Applying the rules transitively. Finally, for enforcing rule number 1 of the definition of NMSI in a transitive manner, it is also necessary to guarantee the following: if $T2$ writes x_{i+1} and y_{j+1}, and subsequently another transaction $T3$ reads y_{j+1} and writes w_k, then the protocol must also prevent $T1$ from reading or overwriting the values written by $T3$, otherwise we would have $T0 \prec T2 \prec T3 \prec T1$, and thus $T1$ should also have read x_{i+1}.

To achieve this, when transaction $T3$ (which read y_{j+1}) commits, for every transaction t that must read version y_l with $l < j+1$ (i.e., the transactions that had entries in the dictionary of y when t_o read y), the protocol will store in the dictionary of every other object w written by $T3$ that t must read the previous version of w (if an older version is not already specified). In the example, since the state for $y.snapshot$ after $T2$ commits specifies that $T1$ must read version y_j, then, when $T3$ commits, $w.snapshot$ is updated to state that $T1$ and $T4$ must read version w_{k-1}.

NMSI rule number 2. Rule number 2 of the NMSI definition says that any pair of transactions that write the same object x must have a relative order, i.e., either $t_i \prec t_j$ or $t_j \prec t_i$. This order is defined by the version number of x created by each transaction.

Therefore, it remains to ensure that this is a partial order (i.e., no cycles). A cycle could appear if two or more transactions concurrently committed a chain of objects in a different order, e.g., if t_m wrote both x_i and y_{j+1} and t_n wrote both x_{i+1} and y_j. To prevent this, it suffices to use a two-phase commit protocol where, for each object, a single accepted prepare can be outstanding at any time.

Waiving SI rule number 3. The fact that NMSI does not have to enforce rule number 3 (which is present only in SI) already allows for some performance gains in the single data center protocol, even though other, more impactful advantages will only become clear when we extend the protocol to multiple data centers in Section 6. In particular, if we consider the example in Figure 1, transaction T_2 is bound to abort in SI after it read version 0 of y concurrently with T_1 creating version 1 of x. This is true of any concurrency control scheme that implements SI because, when T_2 subsequently reads x, SI requires it to return the version corresponding to the snapshot taken when the transaction started (i.e., x_0), and the subsequent write to x would generate a write-write conflict. In contrast, in our NMSI design, the commit of T_1 records in $x.snapshot$ that T_2 should read version 1, which avoids this situation.

5.2 Protocol design

The single data center concurrency control module consists of the following three components: the client library and the transaction managers (TM), which are non-replicated components that act as a front end providing the system interface and implementing the client side of the transaction processing protocol, respectively; and the data managers (DM), which are the replicated components that manage the information associated with data objects.

Client Library. This provides the interface of Blotter, namely *begin*, *read*, *write*, and *commit*. The *begin*, and *write* operations are local to the client. *Read* operations are relayed to the TM, who returns the values and metadata for the objects that were read. The written values are buffered by the client library and only sent to the TM at *commit* time, together with the accumulated metadata for the objects that were read. This metadata is used to set the versions that running transactions must access, as explained next.

Transaction Manager (TM). The TM handles the two operations received from the clients: *read* and *commit*. For *read*s, it merely relays the request and reply to or from the Data Manager (DM) responsible for the object being read. Upon receiving a *commit* request, the TM acts as a coordinator of a two-phase commit (2PC) protocol to enforce the all-or-nothing atomicity property. The first phase sends a *dm-prewrite-request*, with the newly written values, to all DMs storing written objects. Each DM verifies if the write complies with NMSI. If none of the DMs identifies a violation, the TM sends the DMs a *dm-write* message containing the metadata with snapshot information aggregated from all replies to the first phase; otherwise it sends a *dm-abort*.

Data Manager (DM). The core of the concurrency control logic is implemented by the DM. Algorithm 1 describes its handlers for the three types of requests.

(i) Read operation. The handler for a read of object x by transaction T returns either the version of x stored in $x.snapshot$ for T, if the information is present, or the most recent version and then sets $x.snapshot[T]$ to that version, so that a subsequent read to the same variable reads from the same snapshot (and also, as we will see, for propagating snapshot information to enforce NMSI).

Before returning, the read operation must wait in case a concurrent transaction is trying to commit a new value for x (a standard 2PC check, which is done by inspecting x.prewrite). However, blocking is not needed when the snapshot variable forces a transaction to read a prior version.

Algorithm 1: Single data center DM protocols

```
   // read operation
1  upon ⟨ dm-read, T, x ⟩ from TM do
2      processRead ⟨ T, x, TM ⟩;
   // prewrite operation
3  upon ⟨ dm-prewrite, T, x, value ⟩ from TM do
4      if x.prewrite ≠ ⊥ then
           // another prewrite is pending
5          x.pending ← x.pending ∪ {(T, x, value, TM)};
6      else
7          processPrewrite ⟨ T, x, value, TM ⟩;
   // write operation
8  upon ⟨ dm-write, T, x, agg-startd-before ⟩ from TM do
9      for each T' in agg-startd-before do
10         if T' not in x.snapshot then
11             x.snapshot[T'] ← x.last;
12     x.last ← x.last + 1;
13     x.value[x.last] ← x.nextvalue;
14     finishWrite ⟨ T, x, TM ⟩;
   // abort operation
15 upon ⟨ dm-abort, T, x ⟩ from TM do
16     finishWrite ⟨ T, x, TM ⟩;
   // process read operation
17 processRead ⟨ T, x, TM ⟩
18     if T ∉ x.snapshot then
19         if x.prewrite ≠ ⊥ then
20             x.buffered ← x.buffered ∪ {(T, TM)};
21             return
22         else
23             x.snapshot[T] ← x.last;
24     version ← x.snapshot[T];
25     value ← x.value[version];
26     send ⟨ read-response, T, value, {T'|x.snapshot[T'] < version} ⟩ to
         TM;
   // process dm-prewrite request
27 processPrewrite ⟨ T, x, value, TM ⟩
28     if x.snapshot[T] ≠ ⊥ ∧ x.snapshot[T] < x.last then
           // there is a write-write conflict
29         send ⟨ prewrite-response, reject, ⊥ ⟩ to TM;
30     else
31         x.prewrite ← T;
32         x.nextvalue ← value;
33         send ⟨ prewrite-response, accept, {T'|T' ∈ x.snapshot} ⟩ to TM;
   // clean prewrite information and serve buffered
      reads and pending prewrites
34 finishWrite ⟨ T, x, TM⟩
35     if x.prewrite = T then
36         x.nextvalue ← ⊥; x.prewrite ← ⊥;
37     for each (T, TM) in x.buffered do
38         processRead ⟨ T, x, TM ⟩;
39     if x.pending ≠ ⊥ then
40         (T, x, value, TM) ← removeFirst(x.pending);
41         processPrewrite ⟨ T, x, value, TM ⟩;
```

Finally, the metadata returned to the TM are the identifiers of all transactions present in $x.snapshot$ that must read from a version prior to the one returned, i.e., all T' that must obey $T \not\prec T'$ due to T reading x. This information is aggregated by the client library, and propagated to DMs in phase 2 of the commit, to ensure that those T' read a state prior to T. As explained in Section 5.1, this enforces point 1 of the NMSI definition transitively.

(ii) Prewrite operation. This first phase for the commit of T has two goals: detect write-write conflicts, and collect information about concurrent transactions, which is subsequently added to their *snapshot* variables. After checking (and if needed blocking) if there is a concurrent prewrite for an object written by T, the DM detects write-write conflicts by checking if T has to read an older version for any written object x (i.e., by checking $x.snapshot[T]$). If so, then T is writing to an object that was written by a concurrent transaction (i.e., a write-write conflict would violate Rule 1 of the NMSI Definition), and a *reject* is replied. Otherwise, *prewrite* and *nextvalue* are set, in order to block concurrent accesses to x, and an *accept* is returned, including as metadata the identifiers of all transactions in $x.snapshot$. These are the transactions that cannot be serialized after T according to $\prec$, since T is overwriting data they either read or must read. This information is aggregated by the TM and used in the next phase.

(iii) Write and abort operations. If any of the participating DMs detects a write-write conflict, then the TM sends *abort* messages to all DMs involved in T, who then remove T from *prewrite* and *nextvalue*, thus unblocking pending transactions. Otherwise, the TM sends a *write* to all DMs in T, containing the aggregated metadata, comprising the set of identifiers of all transactions present in the snapshot variable for any object read or written by T. Upon receiving a write request, the DM responsible for x will first set $x.snapshot[T']$ to the current version of x, for any transaction T' in the previous set, thus enforcing that $T \not\prec T'$. After this step, the DM will create a new version of x by incrementing its version number, and the new version is made visible to other transactions by updating x.last and x.value[x.last]. Finally, reads pending on that transaction commit are executed, followed by pending prewrites.

5.3 Garbage Collection

The per-object $x.snapshot$ data structure needs a garbage collection mechanism to prevent the number of entries from growing without bound. This is particularly important since these entries are propagated from one data object to another at the time of commit.

An entry $x.snapshot[T]$ guarantees that T reads a version of x that will not break NMSI rules and also enables the detection of write-write conflicts between T and other committed transactions when T attempts to commit. This means that an entry for T in the $snapshot$ data structure only needs to be maintained while T is executing. When T terminates, the entry should be removed as soon as possible to avoid a needless propagation to the $snapshot$ structure of other objects. We take advantage of this observation to implement a simple garbage collection scheme, where each transaction T is created with a time to live (TTL), which reflects the maximum time the transaction is allowed to run. After the TTL of T expires, any entries for T are automatically garbage collected. We analyze the efficiency of this mechanism in Section 7.5.

This mechanism also enables us to garbage collect old versions of data objects: any version of an object x, other than the most recent one, with no entries in the $x.snapshot$ data structure pointing to it can be safely garbage collected.

6. GEO-REPLICATION

Blotter implements geo-replication, with each object replicated in all data centers, using Paxos-based state machine replication [19, 26]. In this model, all replicas execute a set of client-issued commands according to a total order, thus following the same sequence of states and producing the same sequence of responses. We view each data center as a state machine replica. The state is composed by the database (i.e., all data objects and associated metadata), and the state machine commands are the `tm-read` and the `tm-commit` of the TM-DM interface.

Despite being correct, this approach has three drawbacks: (1) read operations in our concurrency control protocol are state machine commands that mutate the state, thus requiring an expensive consensus round; (2) the total order of the state machine precludes the concurrent execution of two commits, even for transactions that do not conflict; and (3) each Paxos-based state machine command

requires several cross-data center message delays (depending on the variant of Paxos used). We next refine our design by addressing each of these concerns, including a discussion on deadlocks.

(1) Local read operations. Read operations modify the snapshot variable state and need to be executed in the state machine, incurring in the cross-data center latency of the replication protocol.

To avoid this overhead, we propose to remove the contents of the snapshot variable from the state machine. The replicas still maintain their view of the snapshot variable, but the information is independently maintained by each replica. This is feasible under NMSI since the snapshot information is used for only two purposes.

The first one is to determine whether write-write conflicts exist. This happens if, at commit time of transaction T, for some modified object x, the snapshot information for T is not the most recent version of x. Since the snapshot information is now updated outside of the state machine, only the replica in the data center where T is initiated records the information for snapshot$[T]$. To allow all data centers to deterministically check for write-write conflicts, we include, as a parameter of the *commit* state machine command, the snapshot information present at the data center where T executed, for each object x modified by T. While this works seamlessly for objects that are both read and written by T, this does not handle the case of a blind write to x, since x.snapshot$[T]$ was not defined in this case. This can be handled by forcing blind writes to perform an artificial read to define x.snapshot$[T]$, right before T commits.

The second use of the information in the snapshot data structure is during transaction execution (i.e., before the commit) to determine which version of an object should be observed by a transaction T, so that a consistent snapshot for T is read. However, this is a local use of the information, and therefore it does not have to be maintained by the state machine. (This forces clients to restart ongoing transactions when failing over to another data center.)

By having consistent reads outside the state machine, read-only transactions can run locally in the data center where the TM runs.

Connection to NMSI. These modifications are possible because, unlike SI, NMSI allows for independence between the state reflected by transactions and the real-time instant when transactions begin and commit. Otherwise, the local snapshot variable might not be up-to-date as other transactions commit (see point (3)).

(2) Concurrent execution of database operations. Enforcing a state machine total order on database operations and executing them serially ensures consistency, but defeats the purpose of concurrency control, which is to enable transactions to execute concurrently.

To address this, we observe that the partial order required by the NMSI definition can be built by serializing the *dm-prewrite* operation on a per-object basis, instead of serializing *tm-commit*s across all objects. This is because a per-object serialization of the first phase of the two-phase commit suffices to ensure that the transaction outcome is the same at all data centers and that the state transformation of each object involved in the *tm-commit* operation is also the same at all replicas, and therefore the database evolves consistently and obeying NMSI across all data centers.

As such, instead of having one large state machine whose state is defined by the entire database, we can have one state machine per object, with the state being the object (including its metadata), and supporting only the *dm-prewrite* operation. The data center where the transaction executes is the one to issue the *dm-prewrite*s for the objects involved in the transaction, and, since the outcome is the same at all data centers, each data center can subsequently commit or abort the transaction independently, without coordination.

Connection to NMSI. This important optimization is made possible by the NMSI semantics, namely the possibility of having long forks. The intersection property between read and write quorums is only required for quorums of the same instance. As such, a commit to an object x may not be reflected in the state read by the Paxos instance for object y and vice-versa, thus leading to a long fork.

Deadlock Resolution. As presented so far, Blotter can incur in deadlocks when more than one transaction writes to the same set of objects, and the Paxos instances for these objects serialize the transactions in a different order. The possibility of deadlock is common across any two-phase commit-based system [28, 9, 23], and, since it has been addressed in the literature, we consider it orthogonal to our contribution. In particular, due to the use of Blotter Paxos, deadlocks are *replicated* across different data centers, and therefore any deterministic deadlock resolution scheme, such as edge-chasing [11], can be employed.

(3) Paxos with a single cross-data center round-trip. We adjusted the configuration of Paxos to reduce the cross data center steps to a single round-trip (from the client of the protocol, i.e., the TM) for update transactions. We leverage two techniques.

The first is to use a variant called Multi-Paxos [19], which allows, in the normal case, for command execution to proceed as follows: client (i.e., TM) to Paxos leader; Paxos leader to all replicas; all replicas to client. The second technique leverages the observation that data center outages are rare, and given that we are using a lower layer of replication protocols to make each Paxos replica fault-tolerant, it is sensible to configure Paxos to only tolerate one unplanned outage of a data center. In fact, this configuration is common in existing deployed systems [2, 9]. (Planned outages are handled by reconfiguring the Paxos membership [20].) Given this observation, we can parameterize Paxos to use read quorums of $N-1$ and write quorums of 2 processes (where N is the number of data centers). This allows the following optimization: upon receiving an operation from the leader, a replica knows that the operation is decided, since it gathered a quorum of two processes between itself and the leader. This allows a TM to commit a transaction incurring in a single cross-data center round trip for each object, irrespectively of the TM being co-located with the Paxos leaders.

Connection to NMSI. This optimization could be applied to other systems that use Paxos in the context of replicated transactions, although, as stated previously, a design that uses different replicas groups for different objects would require quorum intersection across replica groups in SI but not in NMSI. Such quorum intersection would be possible if all groups used the same quorums (e.g., majorities), whereas in NMSI we can use asymmetric quorums and optimize the location of the Paxos leader per-object.

7. EVALUATION

We evaluated Blotter on EC2, by comparing it to Cassandra, an implementation of Spanner's Two-Phase Locking (2PL), a full replication SI protocol [12], and Jessy.

Our evaluation uses various benchmarks and workloads, namely: microbenchmarks for latency and throughput (§7.2); an adaptation of the RUBiS benchmark to key-value stores (§7.3); and a social networking workload (§7.4). We also evaluate the garbage collection mechanism (§7.5); and conduct a separate comparison to Jessy, the other system that offers NMSI (§7.6).

7.1 Experimental Setup

We conducted the experiments on EC2 using data centers from three availability regions: Ireland (EU), Virginia (US-E), and California (US-W). The following table shows the roundtrip latencies between these data centers.

	Ireland	Virginia
Virginia	97	-
California	167	79

A server cluster composed of four virtual machines was setup in each data center. Four additional virtual machines per data center were used as clients. Each virtual machine is an extra large instance with a 64-bit processor architecture, 4 virtual cores, and 15 GB of RAM. Within each data center, keys are mapped to servers using consistent hashing.

We compared the following geo-replicated systems. (In all systems, centralized components, namely lock servers for 2PL and Paxos leaders, ran in Ireland.)

Blotter. We implemented Blotter on top of Cassandra. In particular, we extended Cassandra and its Thrift API with the TM and DM logic of Blotter and implemented a client library supporting the *begin*, *read*, *write*, *commit* interface. For intra-data center replication, Blotter uses Cassandra replication with $N = 2$ replicas, with write quorums of two replicas and read quorums of one replica.

Cassandra. Cassandra is a popular open source NoSQL key-value store [17]. Cassandra does not support transactions, and the consistency of individual operations is defined by the clients, who specify the number of replicas that are contacted in the foreground, before the operation returns. We used two different configurations, both with $N = 3$ replicas (one per data center): (1) the *local quorum* configuration uses read and write quorums of a single replica, thus providing weak consistency; and (2) the *'each' quorum* configuration, where a read operation completes after contacting exactly one replica (at the local data center), and a write operation completes after contacting all three replicas, thus providing strong consistency. Although Cassandra does not support transactions, we compare it against the other systems by clustering its operations into logical groups (which we simply call *transactions*), each containing zero or more read operations and optionally terminated by an aggregated set of write operations.

Spanner's 2PL. We chose Spanner [9] as one of the comparison points because of its relevance, since it is a production system deployed at Google. Update transactions in Spanner are an implementation of the two-phase locking/two-phase commit (2PL/2PC) technique for serializable transactions [4], on top of a Paxos-replicated log. By further leveraging the TrueTime API, Spanner also provides external consistency, or linearizability. (Conversely, Blotter uses Paxos to replicate the atomic commit operation across data centers instead of the log.) We extended Cassandra with transactions using the 2PL/2PC approach of Spanner, with a centralized lock server, on top of a Paxos-replicated log. We left out TrueTime, thus providing serializability, and favoring the performance of our implementation of Spanner's 2PL/2PC.

Generalized SI. Generalized SI (GSI) [12] is an extension of SI suitable for replicated databases. GSI allows transactions to read from *older* snapshots, whereas SI requires transactions to read from the most recent snapshot. The GSI algorithm assumes a full replication scenario where each replica contains the full copy of the database. Read operations can read from any replica, and commit operations must be serialized either using either a centralized transaction certifier or a state-machine approach. (We reused the Paxos implementation of Blotter.) We implemented GSI as an extension of Cassandra, with a replica of each data item per data center, and therefore local reads do not contact remote data centers.

7.2 Microbenchmarks

We measured latency and throughput under a simple workload, which parameterizes the number of read and write operations in each transaction. For this set of experiments, we loaded the database with 10 million random keys and random 256-byte values.

Latency. We first studied how the operations that comprise a transaction affect its latency. In this experiment, each transaction was composed of a single read operation and by either one or five write operations applied at commit time. For each run, the load consisted of a single client machine with a single thread executing 10,000 transactions serially. We used both a single data center in Ireland, and a configuration using all data centers.

The results in Table 1 show the latency (median and 99th percentile) for individual read and commit operations with both one and five write operations (W=1 and W=5).

The single data center configuration evaluates the protocol overheads when the latency between nodes is small. The results show that reads in Blotter incur a slight overhead with respect to the baseline Cassandra implementation (less than half a millisecond) due to the extra steps of writing snapshot information and acquiring locks. For commits, the differences are more pronounced, due the fact that Cassandra only requires two message exchanges, while Blotter, GSI, and 2PL require four because of 2PC.

For the multi-data center configuration, the previous overheads are dwarfed by the inter-data center latency. The Cassandra local quorum configuration is the only one that does not require cross-data center communication, and thus performs similarly in both configurations. Compared to the remaining systems, Blotter and GSI perform better because they only require a response from a single remote data center, in most cases the closest one. Cassandra ('each' quorum) requires replies from all data centers, so the commit latency reflects the latency between the client and the farthest data center. The latency of 2PL is higher than the other protocols because it requires more cross-data center round-trips for commits, and it also has to contact the data center responsible for the read and write locks for both types of transactions.

The results also show that GSI has a slightly lower latency than Blotter, which is likely due to the performance variance of the virtualized, wide-area environment.

Throughput. For measuring throughput, we used transactions with different combinations of read and write operations, including read-only, write-only, and mixed transactions. Each transaction executes R read operations serially followed by a commit with W write operations. We varied the number of concurrent client threads from 12 to 360, equally split across all data centers, where each client thread executes its transactions in a serial order. Figure 3 presents the maximum observed throughput for a configuration spanning all three data centers.

For read-only workloads, 2PL has a much lower throughput due to the fact that is does not necessarily read from the local data center. The overhead due to the extra processing in Blotter compared to Cassandra is also visible, particularly for R=5. For the write-only and mixed workloads, and focusing on the systems that require cross-data center coordination, Blotter has the highest throughput due to its more efficient cross-data center communication pattern and, compared to GSI, because of the increased parallelism, which, as we explained, is fundamentally tied to the use of NMSI.

7.3 Auction Site

We also evaluated Blotter using the RUBiS benchmark, which models an auction site similar to eBay. We ported the benchmark from using a relational database as the storage backend to using a key-value store. Each row of the relational database is stored with a key formed by the name of the table and the value of the primary key. We additionally store data for supporting efficient queries (namely indexes and foreign keys).

The workload consists of a mix with 85% of the interactions containing only read-only transactions, and 15% of the interactions containing read-write transactions. We initially load the key-value store with 10,000 users, 1,000 old items, and 32,667 active items.

System	Single Data Center			Multi Data Center		
	Read Latency	Commit Latency		Read Latency	Commit Latency	
		$W = 1$	$W = 5$		$W = 1$	$W = 5$
Cassandra (local quorum)	0.61 / 4.2	0.62 / 3.3	0.62 / 3.3	1.02 / 6.7	3.41 / 6.3	3.07 / 6.0
Cassandra ('each' quorum)	0.65 / 4.2	1.55 / 3.3	1.51 / 3.3	1.12 / 6.0	180.75 / 189.7	171.49 / 181.7
Blotter	0.98 / 5.0	1.46 / 4.3	1.45 / 4.3	1.60 / 7.5	85.47 / 150.0	87.09 / 128.7
GSI	0.71 / 4.3	1.75 / 5.3	1.76 / 5.3	1.21 / 79.0	79.21 / 82.7	78.89 / 82.3
2PL	0.62 / 4.0	0.61 / 4.0	0.60 / 4.0	1.72 (local), 85.19 (remote) / 150.7	247.32 / 321.7	246.17 / 310.0

Table 1: Latency of microbenchmarks in milliseconds (median / 99th percentile)

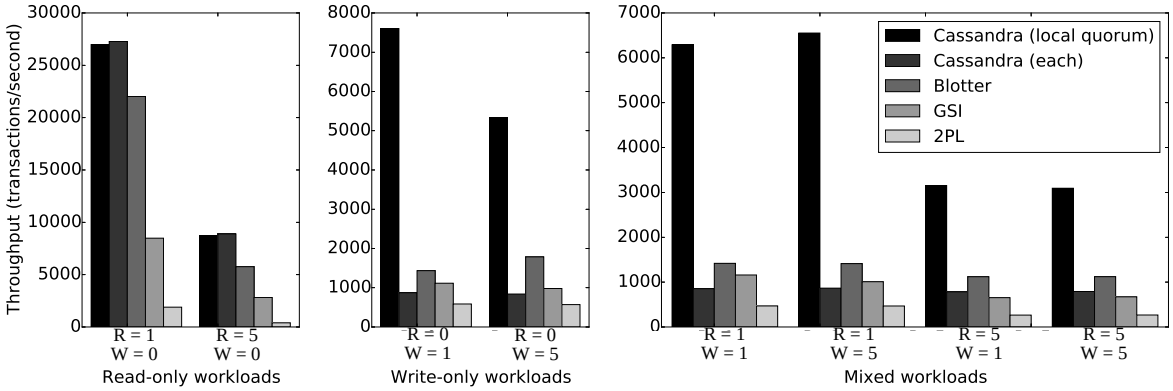


Figure 3: Throughput of multi-data center microbenchmarks

Figure 4 depicts our experimental results for the RUBiS workload. The results show that, consistently with the microbenchmark results, Blotter outperforms 2PL in terms of throughput for both read-only and read-write operations, since the design of Blotter minimizes the need for communication and coordination across data centers.

7.4 Microblogging

This experiment evaluates a mockup implementation of Twitter, supporting three different user interactions, modeled after [30]: *Post-tweet* appends a tweet to the wall of a user and its followers, which results in a transaction with many reads and writes. *Follow-user* appends new information about the set of followers to the profiles of the follower and the followee, which results in a transaction with two reads and two writes. Finally, *read-timeline* reads the wall of the user, resulting in a single read operation.

The workload consists of the following mix of interactions: 85% read-timeline, 10% post-tweet, and 5% follow-user. The database contains 100,000 users and each has an average of 6 followers. For each system, we varied the number of client threads from 120 to 720 and measured the maximum observed throughput.

The results in Figure 5 show a similar pattern to the throughput microbenchmarks. For the read-timeline (read-only) operation, Cassandra achieves the best throughput (60K tx/s), followed by Blotter (50K tx/s), and 2PL (10K tx/s). For the post-tweet and follow-user operations, which contain updates, Blotter has the highest throughput, followed by Cassandra, and then 2PL.

7.5 Garbage Collection

The garbage collection mechanism of Blotter is required to prevent the number of entries in the snapshot data structure of objects from growing without bound, which is important since this impacts the protocol message size.

To analyze the overhead of garbage collection, we deploy Blotter with one server and one client, and set the TTL to 1 second. The client executes a workload parameterized by (1) the number of objects in the database, and (2) the number of read and write operations per transaction. Both directly affect the contention and, consequently, the rate of propagation of snapshot entries in the database. As a metric of the efficiency of the garbage collection mechanism we use the number of snapshot entries returned with each read operation, as this reflects the size of the snapshot data structure at the time of the operation.

Figure 6 shows the average number of snapshot entries per read as a function of how far into the trace we are, for two database configurations: with 1000 objects (less contention) and 100 objects (more contention). For every tested combination of parameters, the GC mechanism stabilizes the size of the snapshot data structures to a very reasonable level of at most a few tens of entries. We also observed a peak in the size of the state that is returned, which happens because the high contention eventually saturates the system, thus increasing the latency of the transactions and lowering the rate at which snapshot entries are propagated.

7.6 Comparison to Jessy

We compared Blotter to the Jessy implementation of NMSI provided by its authors [25]. Jessy was the first system to explore NMSI as the isolation level for transactions in a geo-replicated setting. In contrast to Blotter, it focuses on partial replication settings (i.e., a given data center may not replicate the entire data set). For building consistent snapshots, Jessy uses a data structure called *dependence vector*, which contains either one entry per data item or per set of data items, depending on the configuration. This data structure is stored with data objects in the database, and must be read by clients and propagated within write operations. The propagation of transactions across machines uses total order multicast.

We first ran experiments in a single data center (EC2 in Ireland), where both systems were deployed across four servers, using both dependence vector configurations above. In these experiments we set a replication factor of two and a total of $50,000$

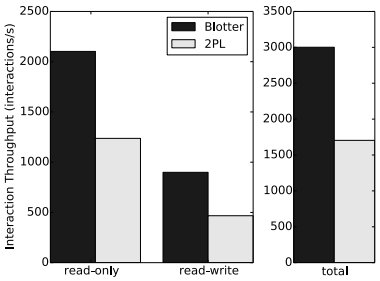


Figure 4: Average user interaction throughput for RUBiS

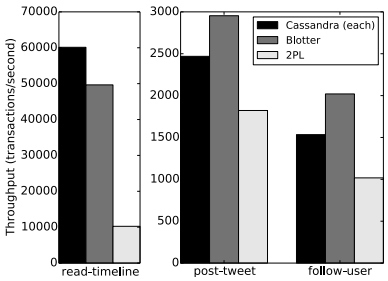


Figure 5: Microblogging throughput

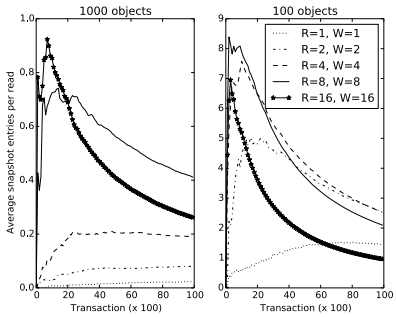


Figure 6: Garbage collection

keys. Each object in this deployment has a size of 256 Bytes. We used YCSB [8] to execute a total of $4,000$ transactions (evenly distributed across 20 client threads). Both configurations of Jessy exhibited a throughput below 100 transactions per second, while the throughput of Blotter was above 300 transactions per second. Even though the performance numbers are not directly comparable, since the code bases are very different, there are several factors that negatively affect the performance of Jessy. In particular, the size of its dependence vectors when using one entry per object grows to an average size of 200 entries only a few seconds into the experiment, which leads to significant memory and communication overheads. In turn, the other configuration of Jessy uses dependence vectors with a constant size of two entries. However, this alternative leads to a significant number of spurious conflicts.

We also deployed Jessy with geo-replication. However, both configurations of Jessy had very low throughput, which is a result using of total order multicast across data centers.

8. RELATED WORK

The closest related proposals are systems that also aim at supporting a non-monotonic version of snapshot isolation transactions in a geo-replicated setting. Jessy [25] was the first system to explore NMSI as the isolation level for transactions in a geo-replicated setting, where objects can be partitioned across different sites. For building consistent snapshots, Jessy uses a data structure called dependence vector, with one entry either per object or per set of objects. The former case has scalability issues, showed by our experiments, since the vector is $O(n)$, where n is the number of objects in the database, whereas in the latter case the algorithm restricts concurrency, leading to spurious conflicts. By focusing on the case where data is fully replicated, Blotter provides consistent snapshots with no spurious conflicts and with an overhead that is linear in the number of active transactions in the local data center.

Walter [28] also uses a non-monotonic variant of SI called parallel snapshot isolation (PSI). While both PSI and NMSI allow long forks, in PSI the snapshot is defined at transaction begin time, based on the state of the replica in which the transaction executes. Ardekani et. al. [24] have shown that this leads to a higher abort rate and results in lower performance when compared to NSMI.

There are also systems that guarantee isolation levels stronger than NMSI in geo-replicated scenarios. The replicated commit protocol [23] provides serializable transactions by layering Paxos on top of two-phase locking and two-phase commit. However, in that protocol, not only commit but also read operations require contacting all data centers and receiving replies from a majority of replicas. In Blotter, we similarly layer Paxos on top of a concurrency control protocol. But, in contrast, Blotter adopts the NMSI isolation level and uses novel concurrency control protocols that allow it to perform consistent reads locally, within the data center where the transaction was started. As a consequence, transactions in Blotter only require a single cross data center round-trip at commit time.

Spanner [9] and Scatter [13] provide strong ACID transactional guarantees with geo-replication, but, in contrast to Blotter, their architecture layers two-phase commit and two-phase locking on top of a Paxos-replicated log. Reversing the order of these two layers leads to more cross data center round-trips and a corresponding drop in performance, as shown by other authors [23].

TAPIR [29] follows the same principle as Blotter of having a lower layer of weakly consistent replication, on top of which transactions are built. However, TAPIR aims for stronger semantics, namely strict serializability. The resulting protocol requires loosely synchronized clocks at the clients (for performance, not correctness), and incurs in a single round-trip to all replicas in all shards that are part of the transaction. In contrast, we offer NMSI without any clock synchronization and with a single round-trip to the closest (or the master) data center.

Other systems also support transactions in a geo-replicated deployment, but provide weaker guarantees than Blotter. MDCC [16] provides read committed isolation without lost updates by combining different variants of Paxos [14]. In contrast to MDCC, Blotter offers stronger semantics, but the ideas of MDCC are complementary since Blotter could make use of a similar approach to further improve the geo-replicated commit.

Some systems like Megastore [3] or SQL Azure [6] provide ACID properties within a partition of data and looser consistency across partitions. In contrast, transactions in Blotter may span any set of objects. Other systems support more limited forms of transactions than Blotter. Eiger [22] supports read-only and write-only transactions. COPS [21] and ChainReaction [1] support read-only transactions that offer causality, but no isolation. In contrast, Blotter supports ACID transactions with NMSI.

9. CONCLUSION

In this paper, we studied the possibility of using NMSI to improve the performance of geo-replicated transactional systems. We proposed Blotter, a system that leverages this isolation level to introduce a set of new protocols. Our evaluation shows that Blotter outperforms other systems with stronger semantics, namely in terms of throughput. As such, our protocols may prove useful for systems that are performance critical and can run under NMSI.

Acknowledgments

Computing resources for this work were provided by an AWS in Education Research Grant. The research of R. Rodrigues is funded by the European Research Council (ERC-2012-StG-307732) and by FCT (UID/CEC/50021/2013). This work was partially supported by NOVA LINCS (UID/CEC/04516/2013) and EU H2020 LightKone project (732505).

10. REFERENCES

[1] S. Almeida, J. Leitão, and L. Rodrigues. Chain- reaction: A causal+ consistent datastore based on chain replication. In *Proc. of 8th European Conference on Computer Systems*, EuroSys'13, pages 85–98, 2013.

[2] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault- tolerant and scalable joining of continuous data streams. In *SIGMOD '13: Proc. of 2013 international conf. on Management of data*, pages 577–588, 2013.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[4] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), January 1981.

[5] N. Bronson et al. Tao: Facebook's distributed data store for the social graph. In *Proc. of the 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.

[6] D. G. Campbell, G. Kakivaya, and N. Ellis. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1021–1024.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[9] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, 2012.

[10] G. DeCandia et al. In *Proc. of the 21st ACM Symposium on Operating Systems Principles*, pages 205–220.

[11] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec.*, 15(3):37–45, Sept. 1986.

[12] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, SRDS '05, pages 73–84, Washington, DC, USA, 2005. IEEE Computer Society.

[13] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, 2011.

[14] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006.

[15] T. Hoff. Latency is everywhere and it costs you sales - how to crush it. Post at the High Scalability blog. http://tinyurl.com/5g8mp2, 2009.

[16] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, 2013.

[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[19] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[20] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, Mar. 2010.

[21] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, 2013.

[23] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[24] M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *Proc. of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS 2013)*, pages 163–172, 2013.

[25] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguiça. On the scalability of snapshot isolation. In *Euro-Par 2013 Parallel Processing*, volume 8097 of *LNCS*, pages 369–381. Springer, 2013.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[27] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, Aug. 2013.

[28] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011.

[29] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proc. of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–278, 2015.

[30] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, SOSP, pages 276–291, 2013.

Exploiting Speculation in Partially Replicated Transactional Data Stores

Zhongmiao Li[†⋆], Peter Van Roy[†] and Paolo Romano[⋆]
[†]Université catholique de Louvain [⋆]Instituto Superior Técnico, Lisboa & INESC-ID

CCS CONCEPTS

• **Information systems → Distributed database transactions**; **Storage replication**;

Online services are often deployed over geographically-scattered data centers (geo-replication), which allows services to be highly available and reduces access latency. On the down side, to provide ACID transactions, global certification (i.e., across data centers) is needed to detect conflicts between concurrent transactions executing at different data centers. The global certification phase reduces throughput because transactions need to hold pre-commit locks, and it increases client-perceived latency because global certification lies in the critical path of transaction execution.

Internal and external speculation. This work investigates the use of two speculative techniques to alleviate the above problems: *speculative reads* and *speculative commits*.

Speculative reads allow transactions to observe the data item versions produced by pre-committed transactions, instead of blocking until they are committed or aborted. Speculative reads can reduce the effective duration of pre-commit locks, thus increasing throughput and reducing latency. Speculative reads are a form of *internal speculation*, as misspeculations never surface to clients.

Speculative commits remove the global certification phase from the critical path of transaction execution, which can further reduce user-perceived latency. Speculative commits are a form of *external speculation*, since they expose to clients the results produced by transactions still undergoing global certification. Thus, speculative commits require programmers to define compensation logic to deal explicitly with misspeculations.

Avoiding the pitfalls of speculation. Past work has shown that the use of speculative reads and speculative commits [3, 4, 6] can enhance the performance of transactional systems. However, these approaches suffer from several limitations:

1. Unfit for geo-distribution/partial replication. Some existing works in this area were not designed for partially replicated geo-distributed data stores, as they either target full replication [6] or rely on a centralized sequencer that imposes prohibitive costs in WAN environments [4].

2. Subtle concurrency anomalies. Existing geo-distributed transactional data stores that support speculative reads [3] expose applications to anomalies, e.g., data snapshots that reflect only partial updates of transactions or include versions created by conflicting concurrent transactions. Such anomalies can be potentially quite dangerous as they can lead applications to exhibit unexpected behaviors (e.g., crashing or hanging in infinite loops) and externalize erroneous states to clients.

3. Performance robustness. In adverse scenarios (e.g., high contention), the injudicious use of speculative techniques can significantly penalize performance, rather than improving it.

Contributions. We propose Speculative Transaction Replication (STR), a novel speculative transactional protocol for partially replicated geo-distributed data stores [5]. STR avoids the problems of centralization by using loosely synchronized clocks, similar to Clock-SI [1]. STR avoids the concurrency anomalies introduced by speculation by obeying a new concurrency criterion called Speculative Snapshot Isolation (SPSI). In addition to guaranteeing Snapshot Isolation (SI) for *committed transactions* [2], SPSI allows an *executing transaction* to read data item versions committed before it started (as in SI), and to atomically observe the effects of non-conflicting transactions that originated on the same node and pre-committed before it started. Finally, to enhance performance robustness STR employs a lightweight self-tuning mechanism that uses hill climbing based on workload measurements to dynamically adjust the aggressiveness of the speculative mechanisms.

Our evaluation shows that the use of internal speculation yields 6× throughput increase and 10× latency reduction in a fully transparent way. Furthermore, applications that exploit external speculation can achieve a reduction of user-perceived latency by up to 100×. These numbers are achieved for both synthetic and realistic workloads characterized by low inter-data center contention, while the self-tuning mechanism ensures gradual fallback to a standard non-speculative processing mode as contention increases.

ACKNOWLEDGEMENT

This work is partially funded by the H2020 project 732505 LightKone, by FCT via projects UID/CEC/50021/2013 and PTDC/EEIÂŋSCR/1743/2014 and by EACEA award 2012-0030.

REFERENCES

[1] Jiaqing Du et al. 2013. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *SRDS*. IEEE, 173–184.
[2] Sameh Elnikety et al. 2005. Database replication using generalized snapshot isolation. In *SRDS*. IEEE, 73–84.
[3] Goetz Graefe et al. 2013. Controlled lock violation. In *SIGMOD*. ACM, 85–96.
[4] Evan Jones et al. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*. ACM, 603–614.
[5] Zhongmiao Li, Peter Van Roy, and Paolo Romano. 2017. *Speculative transaction processing in geo-replicated data stores.* Technical Report 2. INESC-ID.
[6] Paolo Romano et al. 2014. On speculative replication of transactional systems. *J. Comput. Syst. Sci.* 80, 1 (Feb. 2014), 257–276.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SoCC '17, September 24–27, 2017, Santa Clara, CA, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5028-0/17/09.
https://doi.org/10.1145/3127479.3132692

2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Multimodal Indexable Encryption for Mobile Cloud-based Applications

Bernardo Ferreira, João Leitão, Henrique Domingos
DI, FCT, Universidade NOVA de Lisboa & NOVA LINCS
{bf, jc.leitao, hj}@fct.unl.pt

***Abstract*—In this paper we propose MIE, a Multimodal Indexable Encryption framework that for the first time allows mobile applications to securely outsource the storage and search of their multimodal data (i.e. data containing multiple media formats) to public clouds with privacy guarantees. MIE is designed as a distributed framework architecture, leveraging on shared cloud repositories that can be accessed simultaneously by multiple users. At its core MIE relies on Distance Preserving Encodings (DPE), a novel family of encoding algorithms with cryptographic properties that we also propose. By applying DPE to multimodal data features, MIE enables high-cost clustering and indexing operations to be handled by cloud servers in a privacy-preserving way. Experiments show that MIE achieves better performance and scalability when compared with the state of art, with measurable impact on mobile resources and battery life.**

I. Introduction

Mobile devices currently permeate everyday life, surpassing the sales of PCs and Laptops by six times [44] and being responsible for more than 70% of multimedia consumption on the Internet [14]. The advent of mobile devices and tablets has changed the way users produce and manipulate data. On the one hand, users now produce larger quantities of multimodal data (i.e. data containing various media formats such as photos, audio, and text) through their mobile devices [22]. On the other hand, data access and sharing is expected to be ubiquitous [13].

Due to resource limitations (computational power, battery life, and storage capacity) and increasingly larger collections of data produced and accessed by users[1], mobile devices have been a key driving factor for cloud computing solutions and the outsourcing of both data storage and processing [44]. This trend is also known as Mobile Cloud Computing [18]. In such solutions, the cloud effectively operates as a natural extension to the limited storage and computational resources of mobile devices. Furthermore, given such large datasets, being able to efficiently search and retrieve relevant subsets of their data becomes of increased importance for users.

However outsourcing to the cloud inherently leads to dependability and privacy challenges, especially when data and computations are sensitive or of critical nature. This is a natural observation as outsourcing data and computations also entails outsourcing control over them [12]. Recent news have proven that users' privacy is not protected when using cloud services [54]. Governments impose increasing pressure on technological companies to disclose users' data and build insecure backdoors [15], [26]. Malicious or simply careless cloud system administrators have been responsible for critical data disclosures [11], [21]. Finally, one also has to consider internet hackers exploiting software and hardware vulnerabilities in the cloud providers' infrastructure [40].

A common approach for dealing with these dependability issues is to use end-to-end encryption schemes, where users' devices are responsible for encrypting all data before sending it to the cloud [4], [42]. However these schemes restrict functionalities available to users, including efficient data sharing and searching operations through the cloud infrastructure. While data sharing can easily be achieved through key distribution [36], searching encrypted data is a non trivial challenge.

The research community has tried to address this challenge by proposing Searchable Symmetric Encryption (SSE) schemes [2], [10], [16], [27], [34], [35], [49], [55]. Originally designed for exact-match searching in text documents, SSE schemes allow querying encrypted data in sub-linear time, by having users index their data (i.e. build a compact dictionary of the data; e.g. with the unique keywords of each text document) and upload both encrypted index and data to the cloud. However extending SSE to richer queries [2], [8], [57] and other media domains [20], [41] has proven challenging. On one hand, indexing computations of multimodal data and rich media types (including images, audio, and video) are too expensive, especially for mobile client devices and considering that training tasks (i.e. clustering and machine learning algorithms) also have to be performed before data can be efficiently indexed [17]. On the other hand, the few existing approaches [2], [8], [41], [57] are still limited to static collections (i.e. data can't be added, updated, or removed dynamically after deployment and initial load of a repository).

On a side note, searching encrypted data in sub-linear time is only possible by revealing some information patterns to adversaries with each query, including if the query has been performed before and which data objects, although encrypted, were returned by it (search and access patterns [16], respectively). This note is important, as it will be leveraged in the core design of our solution as explained next.

In this paper we propose a novel framework to tackle the practical challenges of supporting mobile applications dynamically storing, sharing, and searching multimodal data in public cloud infrastructures while preserving privacy. We call our proposal MIE - Multimodal Indexable Encryption. MIE leverages from two insights: on the one hand, the leakage of search and access patterns has been proven unavoidable

[1] In cloud-backed multimedia storage apps like iCloud Photos: http://www.apple.com/icloud/photos; and Google Photos: https://www.google.com/photos.

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.31

IEEE computer society

in order to search encrypted data in sub-linear time [48]; on the other hand, in practical deployments where many queries are submitted concurrently by multiple users, these patterns are eventually revealed for the entire index space (i.e. for all possible queries). Leveraging these insights, we contrive MIE to reveal information patterns with each add/update operation, instead of each query. This will allow users to dynamically update and search multimodal repositories while securely outsourcing indexing and training computations to the cloud, which we later show to be the heaviest and more unsuitable computations for mobile applications.

To support MIE's operations we propose a novel family of encoding algorithms with cryptographic properties, called DPE - Distance Preserving Encodings. DPE schemes securely encode data while preserving a controllable distance function between plaintexts. By extracting feature-vectors from multimodal data and encoding them with DPE, users are able to outsource training and indexing computations to the cloud in a privacy-preserving way. We formally define DPE and present two efficient implementations: one for dense media types (e.g. images, audio, and video) and another for sparse media (e.g. text). DPE is of particular interest on itself and can be easily integrated in other secure protocols.

We implemented both an Android and Desktop applications on top of our MIE framework for those platforms. These applications, designed to support the storage and search of multimodal data containing text and image formats, are used to experimentally validate MIE's performance, scalability, and battery consumption in mobile devices. Since (as far as we know) MIE is the first endeavor in multimodal encrypted search, we also implemented a recent SSE scheme from the literature [10], extended it to support multimodal searching, and experimentally compared its performance with MIE. Our implementations are open source and publicly available at: https://github.com/bernymac/MIE.

In summary, this paper makes the following contributions:

• We propose an alternative design to dynamically updating and searching encrypted multimodal data that allows the secure outsourcing of training and indexing computations. We call our proposal MIE - Multimodal Indexable Encryption (§V).

• To support MIE's operations we propose a new family of cryptographic primitives that preserve a controllable distance function between plaintexts. We call our proposal DPE - Distance Preserving Encodings (§IV);

• We implement MIE, both for Desktop and Mobile (Android) devices (§VI), and a multimodal SSE scheme based on the recent literature [10], evaluating and comparing both in terms of performance and scalability across different operations (§VII). Real-world datasets and public commercial clouds (Amazon EC2) are used in these experiments.

II. Related Work

Searching encrypted data is currently a hot research topic, with the increasing popularity of storage and computation cloud services and the security issues they bring. In the last decades, relevant advances have been achieved in powerful cryptographic mechanisms that allow generic computations on encrypted data, including Fully Homomorphic Encryption [23] and Oblivious RAM [56]. However such techniques still remain too expensive to be practical: e.g. computing an AES decryption circuit through fully homomorphic encryption is at least 10^9 times slower [23]; while developing an SSE scheme on top of Oblivious-RAM, protecting access patterns, increases query data-transfer overheads by at least 128 times compared to *conventional* SSE, and by at least 1.75 times compared to downloading the entire database with each query [48].

Searchable Symmetric Encryption (SSE) [16] strives for a practical balance between efficiency and security. Originally designed for exact-match search over static collections of text documents of a single user, SSE schemes are able to achieve sub-linear search performance by initially revealing no information regarding the encrypted data and then gradually revealing some information patterns with each search operation [16]. These leaked information patterns include: search patterns, i.e. has this query been issued before, which is leaked by a deterministic hash of the query; and access patterns, i.e. which data objects are returned by each query, which is leaked by the deterministic identifiers of the objects. Extending SSE for dynamic collections, where documents can be added, updated, and deleted at runtime, initially lead to the further disclosure of update patterns [35], [49] (i.e. if new documents share contents with previously stored documents, leaked by deterministic hashes of the new document's keywords).

Recent dynamic SSE schemes were able to overcome the update leakage issue by increasing operational overhead [34], [55] and/or requiring client storage that grows linearly with the number of unique keywords [6], [10], [27]. Recent works also introduced the concept of forward privacy [6], [55], which states that updates should leak no information even when combined with old query tokens. However in practical scenarios with many queries being submitted by multiple users simultaneously, such guarantees can not hold for long periods.

With the exception of [49], dynamic SSE schemes described so far depend on heuristic models, like the Random Oracle model, which may not have secure implementations in practice and that have been highly criticized in recent years [24]. Making them secure under standard security assumptions requires further client processing and largely increased communication overhead [10], [55], turning these solutions unpractical.

Supporting richer query expressiveness in SSE has not been easy to achieve. The first SSE-based schemes for ranked retrieval were either based on insecure cryptographic primitives [57], or required heavy client processing and search time linear with the index size [8]. These SSE schemes also further revealed frequency patterns, i.e. how many times each queried keyword appears in retrieved documents. Hiding this information has only been possible by assuming the existence of a user-controlled cryptographic module in the cloud server, which would perform multi-party computation with the server, besides encrypting the index with an additively-homomorphic encryption scheme [2]. Furthermore these ranked SSE schemes have so far been restricted to static document collections, as

Scheme	Search Time	Update Time	Client Storage	Revocation Size	Query Type	Search Leakage	Update Leakage
Naveed'14 [49]	$O(m/n)$	$O(m/n)$	$O(1)$	–	Text Match	$ID(w), ID(d)$	$ID(w)$
Cash'14 [10]	$O(m/n)$	$O(m/n)$	$O(n)$	$O(m)$	Text Match	$ID(w), ID(d)$	–
Stefanov'14 [55]	$O(m/n + \log m)$	$O(\log^2 N)$	$O(m)$	–	Text Match	$ID(w), ID(d)$	– (forward private)
Cao'14 [8]	$O(n^2)$	$O(n^2)$	$O(1)$	–	Text Ranked	$ID(w), ID(d)$	$ID(w), freq(w)$
Ferreira'15 [20]	$O(m/n)$	$O(m/n)$	$O(1)$	–	Image Ranked	$ID(w), ID(d)$	$ID(w), freq(w)$
MSSE	$O(m/n)$	$O(m/n)$	$O(n)$	$O(m)$	Multimodal	$ID(w), ID(d), freq(w)$	–
Hom-MSSE	$O(m/n)$	$O(m/n)$	$O(n)$	$O(m)$	Multimodal	$ID(w), ID(d)$	–
MIE	$O(m/n)$	$O(m/n)$	$O(1)$	–	Multimodal	$ID(w), ID(d)$	$ID(w), freq(w)$

TABLE I: Overview of average complexities for the literature on SSE, our work (MIE), and two multimodal SSE schemes (MSSE and Hom-MSSE) designed for baseline experimental comparison by extending the recent literature on SSE [10] (more details in the Evaluation Section §VII). The reader should note that although MIE displays the same search and update time complexities as its two multimodal alternatives, it resorts to more efficient cryptographic primitives, resulting in faster operation time in practice (as will be revealed in §VII). Table Legend: n is the number of unique keywords (or similar concept in other medias, e.g. a keypoint in an image), m is the total number of index entries (keywords or other), $|F|$ is the number of data-objects, $ID(w)$ is the deterministic id of a keyword being queried or added to a data-object, $ID(d)$ represents the ids of the data-objects returned by a query (i.e. that contain the queried keyword), and $freq(w)$ is the frequency of a keyword in data-objects being updated or returned by a query.

they depend on pre-computed and immutable ranking scores that would need to be refreshed and re-encrypted with each document addition, update, or removal.

SSE schemes are usually designed for single writer and single reader/searcher scenarios [2], [10], [27], [55]. Some SSE schemes extend this model to support multiple searchers, however it must be a single writer to generate searching tokens for all other users [8], [57]. Searchable encryption in the public key setting (also know as PEKS) [5] allows the opposite: multiple writers can use a public key to write data, but only a single reader can use the respective private key to search that data. In [53], a multi-key searchable encryption scheme supporting multiple writers and searchers was proposed. However this approach is based on bilinear maps on elliptic curves (which are an order of magnitude slower than conventional symmetric cryptography), has linear-time search performance, and although it supports multiple users it does not address user access control and revocation issues.

Besides text documents, SSE-based schemes have also been designed for other media domains such as images [20], [41]. However, the overhead imposed on client devices in text ranked searching is even more noticeable in the context of images, as machine learning tasks (also known as training) are usually required before dense media types (i.e. images, audio, and video) can be indexed [17]. Furthermore, both training and indexing of dense media data are computationally intensive operations. Some of these performance issues were addressed in [20], however this approach was limited to color features in the image domain. Hence, and to the best of our knowledge, this paper presents the first endeavor in supporting encrypted storage and search of multiple media formats simultaneously (i.e. multimodal data) in a practical way, while supporting resource-restricted mobile devices. Table I provides a summary review of the recent literature on SSE and comparison with our approach across multiple distinguishing factors.

III. Technical Overview

In this section we present an overview of MIE and the system and adversary models that we consider. We start with some notations and fundamental concepts: we call **multimodal data-object**, or simply **object**, to an aggregation of data with multiple media formats or modalities (i.e. an object containing text, image, audio, and/or video; examples are annotated images, wikipedia pages, and personal health records [47]); a **repository** is a collection of multimodal data-objects; **features** are characterizations of objects in some particular media type (e.g. the text modality of an object can be characterized by its most relevant textual keywords [43], while the image modality by a set of visual points of interest [3]); **feature-vectors** are vectorial representations of features, describing an object across its multiple modalities. Feature-vectors are essential components to enable efficient search in repositories containing large collections of multimodal objects.

Multimodal searching uses a multimodal object as query for searching in a multimodal repository. Search results are usually obtained for each media format in separate and aggregated through a multimodal merging function [47].

Indexing takes a collection of data-objects and constructs a dictionary describing them under some features (e.g. which keywords appear in each text document) [43]. This dictionary, called index, forms a compressed representation of the data and allows searching in sub-linear time (e.g. searching for a keyword becomes equivalent to one dictionary access, instead of linearly scanning all text documents).

Training tasks are machine learning operations, such as the k-means clustering algorithm [28], used to find homogeneous groups of objects in dense, high-dimensional data [1]. These groups are used to build more compact representations of high-dimensional data-objects. Example: an object-recognition algorithm [3] finds multiple points of interest in an image. Training a collection of such keypoints from different images yields a group of Distinctive Keypoints [50]. Representing all keypoints of an image in a compact way can then be achieved by finding the most similar Distinctive Keypoint of each and building an histogram with their frequencies.

A. System Model and Architecture

In this paper we focus on the challenges inherent to building practical, secure, and searchable cloud-backed multimodal data repositories especially tailored for mobile devices. We consider

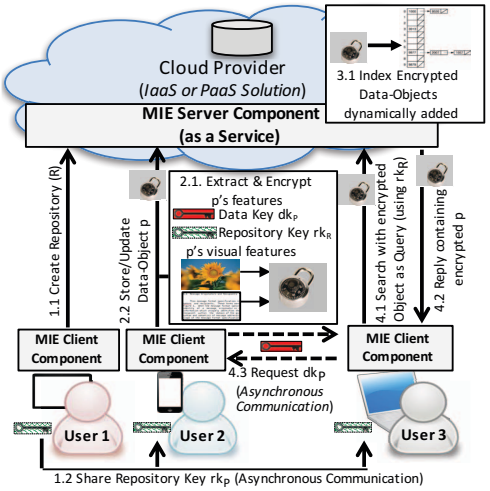


Fig. 1: System model with example interactions between users and the cloud infrastructure, considering image and text media domains.

a system with multiple readers and writers (the **Users**) who store, share, and search data through multiple independent repositories hosted by a **Cloud Server** (or simply **Server**). We assume all data is outsourced to these repositories in the form of data-objects that may contain multiple media formats. A repository is created by one user, and can be used by multiple (authorized) users besides herself. Authorized users can upload their own multimodal data-objects, search through the use of multimodal queries, and retrieve/read objects stored in a repository. Figure 1 provides a high level overview of the described system model.

Upon the creation of a repository, we delegate on the user that created it the task of generating and sharing a **Repository Key** with his trusted users. This cryptographic key allows users to search and add/update objects in that particular repository. More concretely, it is used in the indexing of new/updated objects, as well as in the generation of searching trapdoors. In addition to repository keys we also employ **Data Keys**, used to encrypt the data-objects themselves (using a semantically secure block-cipher, such as AES in CTR mode [36]). Data keys offer users a fine-grained access control over who accesses the full contents of their data-objects; nonetheless they should be seen as an optional functionality, and they can be discarded from the system design in use cases where fine-grained access-control is not required (for instance, by encrypting all data-objects with a shared master key).

When adding (or updating) data-objects in a repository, a user will first process them and extract their feature-vectors in their different modalities. These feature vectors are then encrypted with a Distance Preserving Encoding (DPE, detailed in §IV) and uploaded to the cloud server for training and indexing, alongside the encrypted data-object.

Authorized users with a repository key can also issue multimodal queries, using data-objects with any number of (supported) modalities as queries. To this end they process their query objects the same way as for new data-objects, extracting and encrypting their feature-vectors with DPE and sending them to the cloud server. After receiving an encrypted multimodal query, the cloud server returns the ranked top k matches for it, where k is a configurable parameter. Each of these k matches contains a pair of encrypted data-object and metadata, the later containing deterministic identifiers for the object and its owner (unless data keys have been removed from the system's design, to fully access its contents the querying user will still need to ask the object's owner for its data key).

All remote communications between users and the server should be encrypted and authenticated through secure communication protocols (TLS/SSL [36]). Key sharing interactions can be done asynchronously and out-of-band by resorting to broadcast encryption [16] or a key-sharing protocol based on public-key authentication [36]. User authentication and access control can be achieved through different mechanisms found in the literature, such as sharing authorization tokens between trusted users [16]. This discussion, however, is orthogonal to the main focus of the paper as these mechanisms can easily be integrated into our solution.

B. Adversary Model

In this work we aim at protecting the privacy of users' data and queries. Similar to previous approaches from the literature [2], [10], [16], [27], [55], we consider as main adversary the cloud administrator. This adversary acts in a *honest-but-curious* way, operating the cloud's infrastructure and possibly eavesdropping on users' data, but nonetheless is expected to fulfill its contract agreements and correctly perform operations when asked. We assume that the cloud administrator has access to all data stored on disk or in RAM on any device physically connected to the server, and passing through the network from or to the cloud. Throughout this paper we prove the security of our proposals against such an adversary. We also assume the cloud provider to protect its infrastructure from Internet hackers, as it is in its best interest to protect its infrastructure, its clients, and its reputation.

A stronger adversary that should also be considered is a *malicious user*, i.e. a user of the system who deviates from his expected behavior. Malicious users are an open problem for any multi-user application, as they may be given access to multiple repository and data keys before being discovered, and can more easily eavesdrop on other users' data. In this work we can mitigate the effect of this adversary by providing user access control enforcement and revocation mechanisms, complemented with public-key authentication and periodic key refreshment. Furthermore, we do not consider integrity or availability threats, as they can be handled by different mechanisms orthogonal to the contributions of this paper [38]. Finally, we assume that the higher-level applications using our work can control the amount of background information they reveal, as this may be sensitive and can be leveraged by adversaries for breaking security [9]. In §V-A we discuss possible attack vectors on our work and how to mitigate them.

C. Application Use Case

To provide examples of applications that could benefit from our work, we now briefly discuss a use case and explain the

mapping of concrete entities between it and the previously introduced system model.

Personal Health Records. The number of mobile applications leveraging sensorial data for personal health tracking is growing by a large faction [37]. Moreover, major cloud operators are now offering centralized storage and computation services for such critical health data, under the form of Personal Health Records (PHR) [45]. PHR may contain information regarding users' health conditions under multiple media formats, extracted from their mobile devices' sensors, as well as from medical consultations and healthcare exams performed by healthcare professionals at different medical centers. The availability of this information not only ensures a better healthcare service for patients, but also offers a high potential for the exchange of medical information among different healthcare practitioners and institutes, for medical research purposes and to assist in the treatment of patients with similar conditions.

In this scenario, patients or medical doctors on their behalf (i.e. the Users), outsource their PHR directly from their mobile devices or IoT healthcare devices to a cloud-based backend (i.e. the Cloud Server). Because PHRs belong to the patients, these records can be protected by Data Keys only known to them (and possibly shared with trusted doctors with the patients explicit permission). On the other hand, Repository Keys can be shared between medical doctors and centers, organized in alliance based or medical-specialty based repositories between cooperating professionals. Doctors can then search on these repositories, requesting data keys to PHRs that might be of their interest directly to the respective patients.

IV. Distance-Preserving Encoding

In this section we propose a new family of encoding algorithms, called Distance Preserving Encodings (DPE). Our proposal of DPE comes from the generalization and formal analysis of the main principles behind different existing mechanisms for privacy-preserving nearest-neighbor and similarity computations [7], [20], [33], [41]. DPE is the basis of this work and our new approach to searching multimodal encrypted data. Nonetheless its abstract concept may have interesting applications in other contexts, and as such we present it as an independent building block that doesn't explicitly depend on external aspects of the system using it. We start this section by formally defining DPE. Then we present two efficient implementations of DPE, one applied to dense media types (e.g. images), and another for sparse media (e.g. text). Both implementations are used in §VI to implement an efficient Multimodal Indexable Encryption prototype.

A. DPE Definition

Informally, Distance Preserving Encodings (DPE) are a family of encoding schemes that preserve a controllable distance function between plaintexts, by means of their respective encodings. We say the distance function is controllable, meaning that on instantiation of a DPE scheme a security threshold parameter should be defined, which will allow controlling the amount of information leaked by encodings. More specifically, DPE encodings should only preserve distances between plaintexts up to the value of the threshold. For greater distances, nothing should be leaked by DPE encodings. This threshold allows defining an upper bound on information leakage and security, as it will limit the adversarial ability to perform statistical attacks and establish a distance relation between different plaintexts in the application domain. More formally:

Algorithm 1 Dense-DPE Implementation

function KEYGEN(N, M, Δ)
  $A \leftarrow \mathcal{G}(M \times N)$ ▷ Generate A
  $w \leftarrow \mathcal{G}^{[0,\Delta]}(M)$ ▷ Generate w, limited by 0 and Δ
  $t \leftarrow Func(\Delta)$ ▷ t is controlled by Δ
  return $K = \{A, w\},\ t$
function ENCODE($p, K = \{A, w\}$)
  $e \leftarrow Q(\Delta^{-1}.(A.p + w))$ ▷ $Q(.)$ is fixed
  return e
function DISTANCE(e_1, e_2)
  $D \leftarrow NormHamm(e_1, e_2)$ ▷ Equal to $Eucl(p_1, p_2)$ if $D < t$
  return D

Definition 1 (Distance Preserving Encoding)**.** *A Distance Preserving Encoding (DPE) scheme is a collection of three polynomial-time algorithms (*KEYGEN, ENCODE, DISTANCE*) run by a client and a server, such that:*

- $K, t \leftarrow$ KEYGEN(1^k)*: is a probabilistic key generation algorithm run by the client to setup the scheme. It takes the security parameter k and returns a secret key K and a distance threshold t, both function of and polynomially bounded by k.*
- $e \leftarrow$ ENCODE(K, p)*: is a deterministic algorithm run by the client to encode plaintext p with key K, with p polynomially bounded by k. It outputs an encoding e.*
- $D \leftarrow$ DISTANCE(e_1, e_2)*: is a deterministic algorithm run by the server that takes as input two encodings e_1 and e_2. For plaintext distance function $[0,1] \leftarrow d_p(\cdot,\cdot)$ and encoded distance function $[0,1] \leftarrow d_e(\cdot,\cdot)$ (possibly $d_p = d_e$) with inputs polynomially bounded by k, it outputs $D = d_e(e_1, e_2) = d_p(p_1, p_2)$, if $d_p(p_1, p_2) < t$. Otherwise it outputs $D = t$.*

Given the definition of DPE, in our companion technical report [19] we formally specify all information leaked by its algorithms to an honest-but-curious cloud adversary. We remark that the information leaked is limited to the Distance algorithm and controllable by threshold t. Nonetheless, an adversary can still leverage this leakage to learn some statistics about the data being encoded, and it's up to the applications using DPE to ensure those statistics are not sensitive.

B. A DPE Implementation for Dense Data

Rich media types, including images, audio, and video are characterized by their high-dimensionality and high-density [1]. High dimensionality means that multiple coordinates (the dimensions) are required to describe a point (i.e. a feature-vector) in these media types. As an example, consider the SURF [3] feature extraction algorithm for images, which computes feature-vectors of 64 dimensions. High density means that in all dimensions necessary to describe a feature-vector, most will have a rational value different from zero (even if close, e.g. 0.01). This is defined in clear contrast to sparse

media types such as text, where a document only has a finite subset of keywords from the whole english vocabulary [43] (or any other language) and non-existing keywords can simply be omitted from a feature-vector characterization of the document (e.g. a keyword-frequency histogram).

A DPE implementation for dense data should be able to efficiently encode high-dimensional feature-vectors, while preserving some parametrizable distance function between them. To achieve this goal we extend the encoding proposed by Boufounos et al. [7] for privacy-preserving nearest neighbors. This encoding cryptographically protects feature vectors by transforming them through universal scalar quantization [7]. Moreover, it preserves Euclidean [43] distances between plaintext feature-vectors, through the normalized Hamming [43] distances between encodings, but only up to a tunable threshold t. For plaintext distances greater than t, the distance between encodings conveys no information and will tend to a constant value. More concretely, feature vectors are transformed through the following function:

$$e(x) = Q(\Delta^{-1}(Ax + w)) \tag{1}$$

where $x \in \mathbb{R}^N$ is a N-dimensional feature vector given as input, $A \in \mathbb{R}^{M \times N}$ is a random matrix with independent and identically distributed elements (M is a tunable parameter representing the output size and basically controls the noise introduced by the encoding), Δ is a tunable scaling factor operating element-wise which controls the distance threshold t, $w \in \mathbb{R}^M$ is an additive dither uniformly distributed in $[0, \Delta]$, and $Q(.)$ is a scalar quantizer with non-contiguous intervals such that scalar values in $[2v, 2v+1)$ quantize to 1 and values in $[2v+1, 2v+2)$ quantize to 0, for any v. Finally, $\{A, w\}$ compose the secret key of this scheme.

The previous scheme suffers from a main applicability limitation: secret key $\{A, w\}$ has size proportional to the input and output sizes (N and M respectively). This approach leads to large key sizes and limits flexibility of deployment, as a change on input/output length (e.g. user changes the type of features used for indexing and searching) forces the generation and sharing of a new secret key with the appropriate size. To solve this issue, we introduce a Pseudo-Random Generator (PRG) $\mathcal{G}$ [36] in the key generation algorithm of the previous scheme, instantiated with some random bits of entropy as cryptographic seed. The random values in A and w will be generated through $\mathcal{G}$, and for a Probabilistic Polynomial-Time (PPT) bounded adversary these values are indistinguishable from true random values [36].

Algorithm 1 describes our implementation in detail, which we call Dense-DPE. Consistent with our definition for DPE, Dense-DPE only reveals a distance function between the feature-vectors of data-objects, and this function is limited by threshold t. In our technical report [19] we prove this claim and show that Dense-DPE is a secure realization of DPE.

C. A DPE Implementation for Sparse Data

Since in sparse media types, such as text data, feature-vectors are much smaller compared with dense media types, more efficient algorithms can be used to index and search sparse media. More concretely, to index and search in sparse data, we only need to compare the different non-null values in its feature-vectors for equality[2] (e.g. the keywords of each text document). Translating this to the DPE definition, our DPE implementation for Sparse Data will have a similarity distance threshold of $t = 0$, meaning that it will only reveal if two keywords are equal, and nothing will be revealed even if they are only one character apart.

Algorithm 2 Sparse-DPE Implementation

```
1: function KeyGen(k)
2:     K ← G(k)
3:     t ← 0
4:     return K, t
5: function Encode(p, K)
6:     e ← P_K(p)
7:     return e
8: function Distance(e_1, e_2)
9:     if e_1 == e_2 then
10:        D ← 0
11:    else
12:        D ← 1
13:    return D
```

To achieve the above goals, we base our DPE implementation for Sparse Data on a Pseudo-Random Function (PRF) [36]. More concretely, given a feature-vector from a sparse data-object (i.e. a text document), we apply:

$$f(x) = \mathcal{P}_K(x) \tag{2}$$

where x is a single keyword and $\mathcal{P}$ is a PRF, instantiated with secret key K. In practice, $\mathcal{P}$ can be implemented as a keyed hash function. Algorithm 2 provides the full details of our implementation, which we call Sparse-DPE. In our technical report [19] we prove that Sparse-DPE securely realizes DPE.

V. Multimodal Indexable Encryption

In this section we describe in detail our Multimodal Indexable Encryption (MIE) proposal. The main insight behind MIE is that in practical scenarios where many queries are submitted by multiple users concurrently, the semantic security guarantees initially offered by SSE schemes will not hold for long, as the information patterns leaked with each query will eventually be revealed for the entire index space. However those initial guarantees are only possible by having users train and index their data before uploading it to the cloud, which are heavy operations especially for mobile devices. Leveraging this insight, in MIE we outsource training and indexing computations from user's devices to cloud servers. This is done in a privacy-preserving way by having users extract feature-vectors from the different media formats, encode them with DPE, and upload the encodings to the cloud for computation. The practical result of our approach, on one hand, is that instead of revealing information patterns when queries are performed, as in previous SSE schemes, we reveal them at data creation/update time (namely search, access, and frequency patterns). On the other hand, this approach allows us to

[2]Edit distance and cryptographic schemes such as [33] could be used to construct an alternative Sparse-DPE implementation with threshold distances greater than zero. However, exact string matching complemented with light client-side techniques such as stemming and spell-checking wields similar search precision in ranked text retrieval [43].

effectively support mobile devices dynamically updating and searching multimodal repositories, with increased performance and scalability (see §VII for experimental results).

From a systems perspective, MIE is defined as a distributed framework with two main components: one running in the client device(s), which processes data-objects, extracts feature-vectors in their different modalities, and encrypts them; and another (untrusted) running in the cloud servers, which performs training tasks and indexes data-objects through their encoded features. More formally:

Definition 2 (Multimodal Indexable Encryption)**.** *A Multimodal Indexable Encryption framework is a collection of five polynomial time algorithms (*CREATEREPOSITORY, TRAIN, UPDATE, REMOVE, SEARCH*) executed collaboratively between a user and a server, such that:*

- $\mathbf{rk_R} \leftarrow \mathbf{CreateRepository}(\mathbf{ID_R}, \mathbf{1^{sp_R}}, \{\mathbf{ID_{m_i}}\}_{\mathbf{i=0}}^{\mathbf{n}})$*: is an operation started by the user to initialize a new repository identified by ID_R. It also takes as input a security parameter sp_R and the n modalities to be supported by R ($\{ID_{m_i}\}_{i=0}^{n}$). It creates a repository representation on the server side and outputs a repository key rk_R.*
- $\mathbf{Train}(\mathbf{ID_R}, \mathbf{rk_R}, \{\mathbf{ID_{m_i}}, \mathbf{ip_{m_i}}\}_{\mathbf{i=0}}^{\mathbf{n}})$*: operation invoked by the user to initialize repository R's indexing structures, by performing machine learning tasks (i.e. automatic training procedures), and index its data-objects, if any. The user also inputs the repository key and the indexing algorithms to be used as indexing parameters ($\{ID_{m_i}, ip_{m_i}\}_{i=0}^{n}$, one for each modality; examples of indexing parameters are Inverted List Index and Single Pass In Memory Indexing [43], more details in §VI). This algorithm can be invoked multiple times with different indexing parameters. Note however, that training procedures are only required in dense media types (e.g. images, audio, and video). In a repository containing only sparse media types (e.g. text), this operation will only index existing objects, if any.*
- $\mathbf{Update}(\mathbf{ID_R}, \mathbf{ID_p}, \mathbf{p}, \mathbf{dk_p}, \mathbf{rk_R}, \{\mathbf{ID_{m_i}}\}_{\mathbf{i=0}}^{\mathbf{n}})$*: is the operation used to dynamically add or update a data-object p in repository R. In addition to p, it also takes as input ID_R and ID_p (deterministic identifiers of R and p, respectively), dk_p (data key to be used in the encryption of p), rk_R (repository key of R) and $\{ID_{m_i}\}_{i=0}^{n}$ (the modalities represented in p). If the* TRAIN *algorithm has already been invoked in R, p is indexed in its modalities. Otherwise p's indexing is performed when the* TRAIN *algorithm is invoked for the first time.*
- $\mathbf{Remove}(\mathbf{ID_R}, \mathbf{ID_p})$*: is an operation that allows a user to fully remove a data-object p from repository R and its indexing structures.*
- $\{\mathbf{ID_{p_i}}, \mathbf{p_i}, \mathbf{score^q_{p_i}}\}_{\mathbf{i=0}}^{\mathbf{k}} \leftarrow \mathbf{Search}(\mathbf{ID_R}, \mathbf{q}, \mathbf{rk_R}, \{\mathbf{ID_{m_i}}\}_{\mathbf{i=0}}^{\mathbf{n}}, \mathbf{k})$*: is issued by a user to search in repository R with object q as query, returning the k most relevant data-objects in the repository. Also takes as input the repository key rk_R and the modalities represented in q ($\{ID_{m_i}\}_{i=0}^{n}$). If the* TRAIN *algorithm has been invoked previously for R, the server replies to the query in sub-linear time by accessing R's indexing structures. Otherwise it performs a linear search through R's objects.*

Algorithm 3 Create New Repository

```
1: function USER(U).CREATEREPOSITORY(ID_R, sp_R)
2:     rk1_R ← DENSE-DPE.Keygen(sp_{m_i})
3:     rk2_R ← SPARSE-DPE.Keygen(sp_{m_i})
4:     CLOUD.CreateRepository(ID_R)
5:     RepUsers.ShareKey({rk1_R, rk2_R})
6:     return {rk1_R, rk2_R}
7: procedure CLOUD.CREATEREPOSITORY(ID_R)
8:     Rep[ID_R] ← InitializeRepository()
9:     Fvs[ID_R] ← InitializeFeatureVectorsList()
```

Given MIE's definition, Algorithms 3 through 7 detail our MIE's implementation based on DPE (respectively, operations CreateRepository, Train, Update, Remove, and Search). In our companion technical report [19] we formally specify the information leaked by MIE (which was summarized in Table I) and prove that its DPE-based implementation securely realizes it. In summary, the main difference between our MIE implementation and its secure specification is the use of DPE. Hence, the main argument in proving security lies in showing that by using DPE's algorithms, our MIE implementation doesn't leak anything further to the server.

A. Additional Security Considerations

Applications using MIE have provable security guarantees, equivalent to the ones of previous SSE schemes in practical deployments frequently queried [10], of the information leaked by each operation. However, the impact of this information leakage and to what extent it can be leveraged by adversaries in inference attacks is not yet fully understood. Recent advances have been achieved in this field, with passive [9], [30] and active [58] attacks being proposed, in the text domain, for both query and plaintext recovery. However the efficiency of these attacks depends on very strong assumptions. Passive attacks require almost complete document set knowledge, i.e. adversaries must know the contents of a large subset of all encrypted data. For instance, the best known attack [9] requires 95% document knowledge to achieve 58% query recovery rate. with 75% document knowledge, query recovery drops to values close to 0%. Active attacks can have very strong consequences, but require the adversarial ability of injecting maliciously crafted documents, which must still be encrypted by the client. This means that when deploying a SSE scheme (including MIE), users should control the source of their documents and protect their devices from external hacking.

Regarding other media domains and multimodal data, while keywords in the text domain have a straight semantical meaning, the same may not hold for similar concepts in richer media (including audio, images, and video). Attacks over these domains and their impact are still an open area of research and an interesting future research direction, nonetheless we argue that further background information (controllable by users) may be required for adversaries to achieve acceptable recovery rates in these medias.

VI. IMPLEMENTATION

One of the advantages of our approach lies in its flexibility of deployment and its capacity to integrate different algorithms

Algorithm 4 Train Repository

1: **procedure** USER(U).TRAIN($ID_R, \{rk1_R, rk2_R\}, ID_{m_i}, ip_{m_i}\}_{i=0}^{n}$)
2: CLOUD.Train($ID_R, \{ID_{m_i}, ip_{m_i}\}_{i=0}^{n}$)

3: **procedure** CLOUD.TRAIN(ID_R, $\{ID_{m_i}, ip_{m_i}\}_{i=0}^{n}$)
4: **for all** $\{ID_{m_i}, ip_{m_i}\}_{i=0}^{n}$ **do**
5: $Idx[ID_R][ID_{m_i}] \leftarrow$ InitializeIndex(ID_{m_i}, ip_{m_i})
6: **if** DenseMediaType(ID_{m_i}) **then**
7: $CB_R^{m_i} \leftarrow$ TrainIndex($Idx[ID_R][ID_{m_i}], ip_{m_i}, Fvs[ID_R]$)
8: IndexData($Idx[ID_R][ID_{m_i}], Fvs[ID_R]$)

Algorithm 5 Add/Update Object in Repository

1: **procedure** USER(U).UPDATE(ID_R, ID_p, p, dk_p, $\{rk1_R, rk2_R\}$, $\{ID_{m_i}\}_{i=0}^{n}$)
2: **for all** $\{ID_{m_i}\}_{i=0}^{n}$ **do**
3: $fvs_{m_i}^{p} \leftarrow$ ExtractFeatureVectors(p, ID_{m_i})
4: **if** Dense-Media(ID_{m_i}) **then**
5: $efvs_{m_i}^{p} \leftarrow$ DENSE-DPE.Encode($fvs_{m_i}^{p}, rk1_R$)
6: **else**
7: $efvs_{m_i}^{p} \leftarrow$ SPARSE-DPE.Encode($fvs_{m_i}^{p}, rk2_R$)
8: $e \leftarrow Enc(dk_p, p)$
9: CLOUD.Update($ID_R, ID_p, e, \{ID_{m_i}, efvs_{m_i}^{p}\}_{i=0}^{n}$)

10: **procedure** CLOUD.UPDATE($ID_R, ID_p, e, \{ID_{m_i}, efvs_{m_i}^{p}\}_{i=0}^{n}$)
11: CLOUD.Remove(ID_R, ID_p)
12: $Rep[ID_R][ID_p] \leftarrow e$
13: $Fvs[ID_R][ID_p] \leftarrow \{efvs_{m_i}^{p}\}_{i=0}^{n}$
14: **if** IsTrained(ID_R) **then**
15: **for all** $\{ID_{m_i}\}_{i=0}^{n}$ **do**
16: **for all** $fv \in efvs_{m_i}^{p}$ **do**
17: **if** $Idx[ID_R][ID_{m_i}][fv][ID_p] == \{\}$ **then** $Idx[ID_R][ID_{m_i}][fv][ID_p] \leftarrow 0$
18: $Idx[ID_R][ID_{m_i}][fv][ID_p] ++$

Algorithm 6 Remove Object from Repository

1: **procedure** USER(U).REMOVE(ID_R, ID_p)
2: CLOUD.Remove(ID_R, ID_p)

3: **procedure** CLOUD.REMOVE(ID_R, ID_p)
4: **if** $Rep[ID_R][ID_p]! = \{\}$ **then**
5: $Rep[ID_R][ID_p] \leftarrow \{\}$; $Fvs[ID_R][ID_p] \leftarrow \{\}$
6: **if** IsTrained(ID_R) **then**
7: **for all** $\{ID_{m_i}\}_{i=0}^{n}$ **do**
8: **for all** $fv \in Idx[ID_R][ID_{m_i}]$ **do**
9: $Idx[ID_R][ID_{m_i}][fv]$.Remove(ID_p)

Algorithm 7 Search Repository with Object as Query

1: **function** USER(U).SEARCH($ID_R, q, \{rk1_R, rk2_R\}, \{ID_{m_i}\}_{i=0}^{n}, k$)
2: **for all** $\{ID_{m_i}\}_{i=0}^{n}$ **do**
3: $fvs_{m_i}^{q} \leftarrow$ ExtractFeatureVectors(q, ID_{m_i})
4: **if** Dense-Media(ID_{m_i}) **then**
5: $efvs_{m_i}^{q} \leftarrow$ DENSE-DPE.Encode($fvs_{m_i}^{q}, rk1_R$)
6: **else**
7: $efvs_{m_i}^{q} \leftarrow$ SPARSE-DPE.Encode($fvs_{m_i}^{q}, rk2_R$)
8: $\{ID_{p_i}, p_i, score_{p_i}^{q}\}_{i=0}^{k} \leftarrow$ CLOUD.Search(ID_R, $\{ID_{m_i}, efvs_{m_i}^{q}\}_{i=0}^{n}$, k)
9: **return** $\{ID_{p_i}, p_i, score_{p_i}^{q}\}_{i=0}^{k}$

10: **function** CLOUD.SEARCH(ID_R, $\{ID_{m_i}, efvs_{m_i}^{q}\}_{i=0}^{n}$, k)
11: **for all** $\{ID_{m_i}, efvs_{m_i}^{q}\}_{i=0}^{n}$ **do**
12: **if** IsTrained(ID_R) **then**
13: $hist_{m_i}^{q} \leftarrow$ ClusterizeAndSort($CB_R^{m_i}, fvs_{m_i}^{q}$)
14: $Res_{m_i} \leftarrow Idx[ID_R][ID_{m_i}]$.IndexSearch($hist_{m_i}^{q}$)
15: **else**
16: $Res_{m_i} \leftarrow$ LinearRankedSearch($efvs_{m_i}^{q}, Fvs[ID_R]$)
17: $Res_{m_i} \leftarrow$ Sort(Res_{m_i})
18: $Res \leftarrow$ FusionRank($\{ID_{m_i}, Res_{m_i}\}_{i=0}^{n}, k$)
19: **return** $\{ID_{p_i}, Rep[ID_{p_i}], Res[ID_{p_i}]\}_{i=0}^{k}$

for feature extraction (client side) and both training and indexing computations (server side). MIE is agnostic to the information retrieval techniques used on either side, and they can be used in the encrypted domain without any major modifications from their original plaintext algorithms. With this in mind, we implemented a prototype version of MIE to experimentally validate its design and compare it with the most relevant approaches from the literature. These experimental results are detailed in §VII, while for now we focus on our prototype description. The user-side component of MIE was developed as an Android Service, using a mixture of Java with Android's SDK and C++ with Android's Native Development Kit. The cloud server component was fully developed in C++.

In order to showcase its multimodality, we implemented our prototype supporting text and image data. Text feature extraction on the user's side is performed through standard keyword stemming, stop-words removal, and histogram extraction [43], followed by Sparse-DPE encoding. Regarding image feature extraction, since our Dense-DPE implementation currently preserves Euclidean distances between plaintext feature-vectors, it is more suitable for floating-point image descriptors. As such, we use the *SURF* descriptor extraction algorithm [3] and *Dense Pyramid* feature detection [39] for our prototype implementation. Dense-DPE was instantiated with threshold $t = 0.5$ and output size equal to the input size (64 dimensions for *SURF* feature-vectors). As cryptographic algorithms' implementations, we use HMAC-SHA1 as implementation of Pseudo-Random Functions (PRFs), AES in CTR mode for data-objects encryption, and an AES-based Pseudo-Random Number Generator (PRNG) for random number generation. OpenSSL 1.0.2 [51] and OpenCV 2.4.10 [31] were compiled for Android integration and support MIE's user-side cryptographic and image retrieval computations, respectively (all remaining computations, including text feature-extraction, were implemented by us).

On the server side we use an index per modality, for each repository (as previously discussed in MIE's design). Both for text and image data, the inverted index [43] approach is used, where each index key represents a distinct keyword and index values compose a list of all object identifiers containing the keyword. Since this type of index was originally designed for text data, we use the Bag-Of-Visual-Words (BOVW) model as an intermediary step to represent image features as visual words [50]. In this model, feature-vectors extracted from a repository's images are clustered in a machine-learning step (MIE's training operation), through a clustering algorithm such as k-means [50]. This training step selects a number of representative feature-vectors (1.000 in our experiments) which are called visual words. After this step, when adding/updating or searching, the different feature-vectors of the input image can be matched with the selected visual words, and the most similar ones are used henceforth to represent each feature-vector. This way, the frequency of visual words in an image become similar to the frequency of keywords in text documents. Each visual word is given an index key, and a tree-like structure is built over all visual words, through hierarchical k-means [50], in order to improve visual word comparison performance (we use a visual-words tree of height 3 and width 10).

To further improve scalability, if an index (of any modality)

grows too large to fit in the cloud server's main memory, champion posting lists [43] are used to ensure that only the top ranked data-objects for each index entry are kept in memory, while the full index is stored in disk and periodically merged with updated/newly added index entries. This technique improves scalability without impacting retrieval precision. Again we remark that due to the properties of MIE and DPE, only small modifications are required for these techniques to work in the encrypted domain (such as applying k-means over normalized Hamming distances due to Dense-DPE properties, instead of Euclidean distances as in its original design).

To rank search results, the TF-IDF [43] weighting function is used both for images and text. Nonetheless more complex functions could be used without loss of generality (e.g. BM25 [43]). Finally, to enable multimodal querying (simultaneous search with multiple media query formats) we use the logarithmic inverse square rank fusion approach [46]. This approach allows us to separately search in the different modalities and then merge all obtained results into the final set of multimodal results, according to the rankings in each modality.

Training and k-means computations in the cloud side are done using OpenCV 2.4.10, and all other computations (including indexing and searching) were implemented by us. Once again we remark that the prototype described is one of many information retrieval combinations made possible by MIE's design, and should be seen as a reference implementation. To showcase the potential of our framework, we also implemented a simple Android and desktop applications which exercise all operations provided by MIE.

VII. Experimental Evaluation

In this section we experimentally evaluate MIE, through the prototype implementation described in the previous section. For experimental baseline comparison, we also extended a recent SSE scheme from the literature [10] to support ranked multimodal querying and implemented two variants: one that is a simple extension of its mechanisms and hence leaks search, access, and frequency patterns; and another where the user encrypts the index with an additively-homomorphic encryption scheme [52], protecting frequency patterns when performing queries. We refer to these schemes as MSSE and Hom-MSSE, respectively, and their full implementation details can be found in our companion technical report [19].

Experimental Test-Bench In the following we will present perfomance results for the MIE, MSSE, and Hom-MSSE alternatives, comparing results both from Desktop and Mobile clients and analyzing them to the grain of each sub-operation. As Mobile client device we used a 2013 Nexus 7 Android Tablet, equipped with a Qualcomm Snapdragon S4 Pro quad-core 1.5Ghz CPU, 2 GB RAM running Android Lolipop 5.1.0. As Desktop client we used a Macbook Pro with Mac OS X 10.11, 4GB of RAM, and 2.3Ghz quad-core Core i7 CPU. For the Cloud server, we used an Amazon EC2 m3.large instance, where the average round-trip time for client-server communications is 52.160 ms. In these experiments, the mobile client is connected to the Internet through WIFI 802.11g and the Desktop Client through an ethernet cable (100 mb/s). As dataset we used the MIR-Flickr dataset [29], which contains one million images and their user defined textual tags extracted from the Flickr social network.

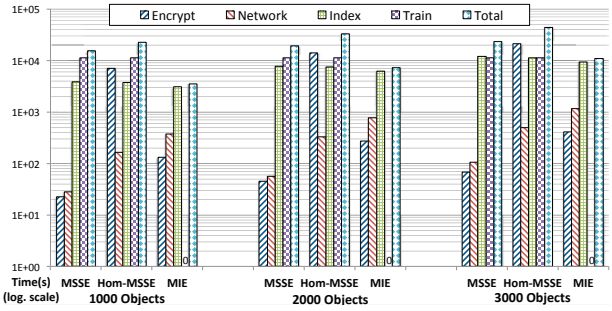


Fig. 2: Performance of the update operation in a mobile device

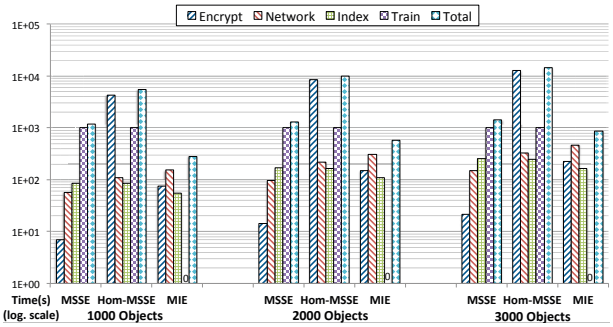


Fig. 3: Performance of the update operation in a desktop device

Experimental Evaluation Roadmap The goal of our experimental work is to answer the following questions: **i**) what are the implications on user perceived performance (i.e, time consumed by user devices) to process and upload multimodal data to a cloud infrastructure, considering different devices (mobile and desktop) and how performance evolves as we scale the size of the data set in a scenario where a single user is accessing the repository (§VII-A)? **ii**) As MIE was designed to support multiple users and facilitate concurrent accesses to repositories, what are the implication on user perceived latency when two clients concurrently add objects to the same repository (§VII-B)? **iii**) What is the user perceived performance associated with searching a repository using MIE and the concurrent schemes (§VII-C)? **iv**) What is the retrieval precision obtained by MIE, in comparison with the concurrent schemes and with plaintext retrieval (§VII-D)? And finally, **v**) what are the implications of the different schemes on the battery life of mobile devices when users upload new multimodal content to a repository, and how this varies as dataset sizes grow (§VII-E)?

A. Single User Scenario

Figures 2 and 3 report the results for the time consumed by respectively, a client executing in a mobile device and in a desktop computer, when initializing a repository and uploading a variable number of multimodal data objects (varying from $1,000$ to $3,000$). Notice that the y-axis in these figures is presented in a logarithmic scale for improved readability. Re-

sults are divided between sub-operations: *Encrypt* represents the performance of encryption operations in the three schemes; *Network* represents the time spent with communications and uploading data to the cloud server; *Index* is the time spent by the client extracting multimodal feature-vectors and indexing them; *Train* is the performance of the training operation, where machine learning tasks are performed; *Total* represents the sum of all sub-operations.

We start by noting that when compared with MSSE and Hom-MSSE, the client that leverages MIE (both in desktop and mobile devices) does not consume any time on the training operation. This is due to MIE's ability to offload this heavy computational step to the cloud in a secure way.

Furthermore the time spent on indexing by MIE clients is lower when compared with MSSE and Hom-MSSE. In this step MIE clients only have to extract feature-vectors from the plaintext data-objects in the different modalities. By encrypting those feature-vectors with our Distance-Preserving schemes (DPE), all other indexing computations are securely offload to the cloud server. In contrast, MSSE and Hom-MSSE clients have to perform those operations in their devices, which include: clustering feature-vectors against the training data-structures obtained during the training step (for dense media types); and indexing those feature-vectors (or their clustered versions), storing the results in encrypted indexing structures which are then uploaded to the cloud.

In the Encryption sub-operation, Hom-MSSE clients exhibit the worst performance due to the use of additively-homomorphic encryption. MIE clients waste more time than MSSE in this sub-operation, as DPE is more expensive than the standard cryptographic primitives used in MSSE, and in the Networking sub-operation MIE clients also show worse performance than the competing schemes, as MIE clients have to upload encoded feature-vectors to the cloud while MSSE and Hom-MSSE only have to upload the already processed and encrypted indexing structures. However, and even if we dismiss the cost of the training operation, MIE clients still show lower total execution time than MSEE and Hom-MSSE clients. The average performance cost increase from MIE to MSEE and Hom-MSSE, considering the three datasets and dismissing training costs, is around 9% and 203% respectively.

Concerning the observed performance across different devices (mobile vs desktop), the relative time spent on each operation for each of the evaluated schemes remains mostly unchanged. However, and as expected, CPU intensive operations such as encryption, indexing, and training perform faster on the desktop computer (approximately 1 order of magnitude). This is explained by the difference in CPU power available in each device. Nonetheless, in both devices and across all data set sizes, MIE allows more efficient processing and storage of multimodal data than the competing alternatives. Consequently, MIE also allows users to initialize and load a secure cloud-backed repository with searchable capabilities in much less time than the competing alternatives (by one order of magnitude approximately). This shows the effectiveness of our alternative, which is able to outsource heavy computational steps to the cloud by exposing at create/update time the same information patterns that the remaining alternatives leak when executing search operations.

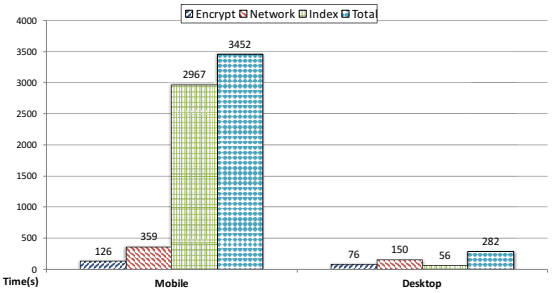


Fig. 4: Multiple client simultaneous update performance, with 1 mobile and 1 desktop client where each upload $1,000$ data-objects

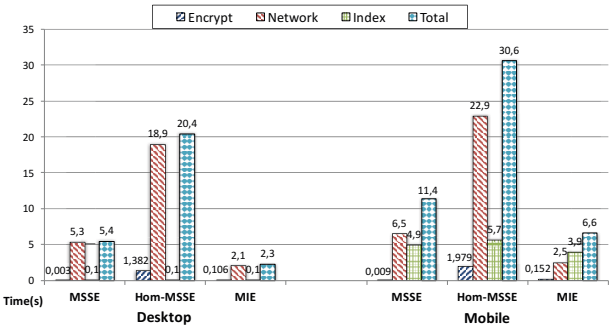


Fig. 5: Performance of the search operation for Mobile and Desktop

B. Multiple Users Scenario

We next conducted an experiment where two clients, one executing in a desktop computer and the other on a mobile device, process and upload $1,000$ multimodal data objects each to a single cloud-backed repository. In this experiment we only evaluated the MIE approach, since MSSE and Hom-MSSE (as well as previous SSE schemes [10]) are not easily extendable to multiple users, as they require client-storage that must be consistently synchronized between all users of a repository[3]. Our MIE approach requires no client storage and was designed to enable concurrent write access to data repositories, hence both clients in the experiment can progress at the same time.

Figure 4 summarizes the results for both clients. The figure shows that when compared with the results of the previous sub-sections, both clients are able to make independent progress and both consume the same amount of time when storing a dataset composed of $1,000$ multimodal objects.

C. Query Performance

Figure 5 reports the total time required by a client (either on a desktop computer or a mobile device) to perform a query on a repository with $1,000$ multimodal objects and obtain an answer from the cloud infrastructure. In this experiment, since searching is a synchronous operation (contrary to the previous

[3]SSE schemes could use some form of strongly consistent distributed storage in order to keep client state synchronized between users, however such an approach, especially on mobile devices, would increase performance and bandwidth overheads even further.

	Plaintext	MSSE	Hom-MSSE	MIE
mAP (%)	57.938	57.965	57.881	57.562

TABLE II: Mean Average Precision (mAP) for the Holidays dataset [32]

operations that were asynchronous), the *Network* sub-operation contemplates the time spent on communications with the cloud servers and the time the cloud servers take to respond to the query. The results show that in both devices MIE out-performs significantly the competing solutions MSSE and Hom-MSSE. The reasons that explain this are two-fold. First, MIE was designed to only extract feature-vectors from the multimodal object used as query, while the other approaches also have to cluster these feature-vectors with the output of the training task, in order to determine the index positions that should be accessed by the cloud servers. The effect of this is shown in the *Index* sub-operation. Second, MIE requires less computational effort in the cloud servers than the MSSE and Hom-MSSE approaches, which is shown in the *Network* sub-operation. As expected, on mobile devices all solutions take more time than in the desktop computer to process and fetch relevant information for a query, however the increase is proportional across the different schemes.

These results clearly show that not only MIE is more performant than MSSE and Hom-MSSE, but it is also well suited for mobile devices when storing multimodal data on a public cloud infrastructure and when performing queries to retrieve data objects.

D. Query Precision

Dense-DPE, used in the encryption of dense feature-vectors (e.g. those extracted from images), is the only MIE component that may possibly introduce entropy for retrieval operations, affecting query results. As such, we assessed the retrieval precision obtained by MIE and the competing alternatives when querying an image-only repository. This evaluation was performed using the Inria Holidays dataset and its evaluation package [32], measuring the mean average precision (mAP) of 500 queries over a repository of 1491 photos. Table II shows an average of 10 independent executions for MIE, the competing alternatives MSSE and Hom-MSSE, and a plaintext retrieval system based on the same image retrieval techniques.

All assessed systems obtained similar retrieval precision results. Dense-DPE (in MIE) does not meaningfully affect retrieval precision as long as encoded features are at least as large their plaintext versions. Homomorphic encryption (in Hom-MSSE) also seems to preserve the precision of the retrieval algorithms. Finally, we believe that the result of the training operation may have a more meaningful impact on retrieval precision than any other component in the framework system, as clustering is a NP-Hard problem and only an approximated solution can be found [28].

E. Mobile Energy Consumption

As one of our goals is to provide adequate support to mobile devices, it is relevant to measure the draining of energy from a mobile device battery when creating a cloud-based repository and loading it with $1,000$, $2,000$, or $3,000$ multimodal objects. We also report the energy required to

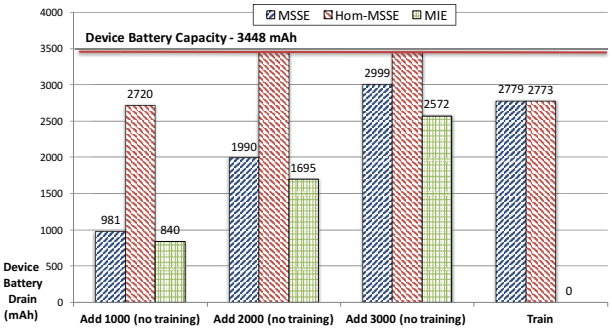


Fig. 6: Mobile energy consumption for the different operations

train the repository using machine learning techniques, which is required by the MSSE and Hom-MSSE solutions. For improved readability, the results for training and adding the three datasets are shown in separate. The measured energy capacity of the battery in the mobile device used in these experiments was $3,448mAh$. Figure 6 reports the obtained results, which were measured through Android's Operating System Power Profiles Framework [25]. This framework allows users to verify in a precise and hardware-backed way how much energy is consumed in a given period of time by the different applications running in the system.

Results show that MIE significantly outperforms the remaining schemes. This is a reflection of the results shown in the previous sub-sections, and further proves that MIE is more lightweight and better suited for mobile adoption than the state of the art alternatives. For the $2,000$ and $3,000$ dataset sizes, the Hom-MSSE scheme surpassed the available energy capacity, causing the mobile device to shutdown before completion of the test. Furthermore, as shown in Figure 6, MIE is also able to avoid the train operation which almost depletes the energy of the mobile device on its own. These results show that MIE is effectively the solution which is best tailored for operation on mobile devices with limited energy life.

VIII. Conclusion

In this paper we have tackled the practical challenges of efficient and dynamic storage and search of encrypted multimodal data on public clouds, while supporting resource constrained mobile devices. Our main contribution, named *Multmimodal Indexable Encryption* (MIE), is the first approach to address this problem, and is particularly suited for practical contexts and mobile devices. At the core of MIE lies a novel family of encoding algorithms, called *Distance Preserving Encoding* (DPE), which preserve a controllable distance function between plaintexts after encoding. By leveraging DPE, MIE is able to outsource indexing and training computations (shown to be the core of heaviest computations) from the mobile devices to the cloud servers in a secure way. We have implemented a prototype of MIE, operating both on desktop computers and Android mobile devices. Our prototype supports both textual and image modalities. We have experimentally shown that MIE is more adequate than other approaches for storing and searching encrypted multimodal data, especially

when client applications are executed in resource constrained mobile devices.

ACKNOWLEDGMENTS

The authors thank Miguel Correia for helpful comments and feedback. This work was supported by FCT/MCTES through project NOVA LINCS (UID/CEC/04516/2013) and the EU, through project LightKone (grant agreement n^o 732505).

REFERENCES

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *SIGMOD'98*, pages 94–105, 1998.
[2] F. Baldimtsi and O. Ohrimenko. Sorting and Searching Behind the Curtain. In *FC'15*, 2015.
[3] H. Bay, T. Tuytelaars, and L. V. Gool. SURF: Speeded Up Robust Features. In *ECCV'06*, pages 404–417. Springer, 2006.
[4] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSKY: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage*, 9(4), 2013.
[5] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Eurocrypt 2004*, pages 506–522. Springer, 2004.
[6] R. Bost. Sophos - Forward Secure Searchable Encryption. In *CCS'16*. ACM, 2016.
[7] P. Boufounos and S. Rane. Secure binary embeddings for privacy preserving nearest neighbors. In *IEEE International Workshop on Information Forensics and Security*, pages 1–6. Ieee, nov 2011.
[8] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-Preserving Multi-Keyword Ranked Search over Encrypted Cloud Data. *IEEE Transactions on Parallel and Distribibuted Systems*, 25(1):222–233, 2014.
[9] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *CCS'15*, pages 668–679. ACM, 2015.
[10] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *NDSS'14*, volume 14, 2014.
[11] A. Chen. GCreep: Google Engineer Stalked Teens, Spied on Chats. http://gawker.com/5637234, 2010.
[12] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *CCSW'09*, 2009.
[13] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016 – 2021. https://tinyurl.com/zzo6766, 2017.
[14] ComScore. The 2016 U.S. Mobile App Report. Technical report, 2016.
[15] T. Cook. A Message to Our Customers. Apple. https://www.apple.com/customer-letter/, 2016.
[16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *CCS'06*, pages 79–88, 2006.
[17] R. Datta, D. Joshi, J. Li, and J. Z. Wang. Image retrieval. *ACM Computing Surveys*, 40(2):1–60, 2008.
[18] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.
[19] B. Ferreira, J. Leitão, and H. Domingos. Multimodal Indexable Encryption for Mobile Cloud-based Applications (Extended Version). Technical Report. Available at: http://asc.di.fct.unl.pt/%7Ebf/MIE.pdf, 2016.
[20] B. Ferreira, J. Rodrigues, J. Leitão, and H. Domingos. Privacy-Preserving Content-Based Image Retrieval in the Cloud. In *SRDS'15*. IEEE, 2015.
[21] T. Frieden. VA will pay $20 million to settle lawsuit over stolen laptop's data. http://tinyurl.com/lg4os9m, 2009.
[22] B. Fung. In 5 years, 80 percent of the whole Internet will be online video. http://tinyurl.com/jxod8kf, 2015.
[23] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO'12*, pages 850–867. Springer, 2012.
[24] S. Goldwasser and Y. T. Kalai. Cryptographic Assumptions: A Position Paper. Cryptology ePrint Archive, Report 2015/907, 2015.
[25] Google. Power Profiles for Android. https://source.android.com/devices/tech/power/index.html.
[26] G. Greenwald and E. MacAskill. NSA Prism program taps in to user data of Apple, Google and others. http://tinyurl.com/oea3g8t, 2013.
[27] F. Hahn and F. Kerschbaum. Searchable Encryption with Secure and Efficient Updates. In *CCS'14*, pages 310–320. ACM, 2014.
[28] J. A. Hartigan. *Clustering algorithms*. Wiley, 1975.
[29] M. J. Huiskes and M. S. Lew. The MIR Flickr Retrieval Evaluation. In *MIR '08*, New York, NY, USA, 2008. ACM.
[30] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.
[31] Itseez. OpenCV: Open Source Computer Vision. http://opencv.org.
[32] H. Jegou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *Computer Vision-ECCV*, pages 304–317. Springer, 2008.
[33] A. Juels and M. Wattenberg. A Fuzzy Commitment Scheme. *CCS '99*, pages 28–36, 1999.
[34] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. *FC'13*, pages 1–15, 2013.
[35] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *CCS'12*, pages 965–976. ACM, 2012.
[36] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC PRESS, 2007.
[37] S. Khalaf. Health and Fitness Apps Finally Take Off, Fueled by Fitness Fanatics. http://tinyurl.com/q4wyl7j, 2014.
[38] B. H. Kim and D. Lie. Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices. *S&P'15*, 2015.
[39] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR'06*, volume 2, pages 2169–2178. IEEE, 2006.
[40] D. Lewis. iCloud Data Breach: Hacking And Celebrity Photos. https://tinyurl.com/nohznmr, 2014.
[41] W. Lu, A. Swaminathan, A. L. Varna, and M. Wu. Enabling Search over Encrypted Multimedia Databases. In *IS&T/SPIE Electronic Imaging*, pages 725418–725418–11, feb 2009.
[42] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Transactions on Computer Systems*, 29(4):1–38, dec 2011.
[43] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*, volume 1. Cambridge University Press, 2009.
[44] M. Meeker. Internet Trends 2015. In *Code Conference*, 2015.
[45] Microsoft. HealthVault. https://www.healthvault.com/, 2016.
[46] A. Mourão, F. Martins, and J. Magalhães. NovaSearch at TREC 2013 Federated Web Search Track : Experiments with rank fusion. In *TREC*, number Task 1, pages 1–8, 2013.
[47] A. Mourão, F. Martins, and J. Magalhães. Multimodal medical information retrieval with unsupervised rank fusion. *Computerized Medical Imaging and Graphics*, may 2014.
[48] M. Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Technical report, Cryptology ePrint Archive, Report 2015/668, 2015.
[49] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic Searchable Encryption via Blind Storage. In *IEEE S&P*, 2014.
[50] D. Nistér, H. Stewénius, D. Nister, and H. Stewenius. Scalable recognition with a vocabulary tree. In *IEEE CVPR'06*, volume 2, pages 2161–2168. IEEE, 2006.
[51] OpenSSL Software Foundation. OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org.
[52] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99*, pages 223–238, 1999.
[53] R. A. Popa, E. Stark, J. Helfer, and S. Valdez. Building web applications on top of encrypted data using Mylar. In *NSDI'14*, 2014.
[54] D. Rushe. Google: don't expect privacy when sending to Gmail. http://tinyurl.com/kjga34x, 2013.
[55] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *NDSS'14*, 2014.
[56] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, S. Devadas, M. V. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. *CCS'13*, 2013.
[57] C. Wang, N. Cao, K. Ren, and W. Lou. Enabling Secure and Efficient Ranked Keyword Search over Outsourced Cloud Data. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1467–1479, aug 2012.
[58] Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *Usenix Security '16*, 2016.

Testing properties of weakly consistent programs with Repliss

Peter Zeller
TU Kaiserslautern, Germany
p_zeller@cs.uni-kl.de

ABSTRACT

Repliss is a tool for the verification of programs which are built on top of weakly consistent databases. As one part of Repliss, we have built an automated, property based testing engine. It explores executions of a given application with randomized invocations and scheduling while checking for invariant violations. When an invariant is broken, Repliss minimizes the execution and displays a visualization of the minimized failing execution. Our contributions are 1. heuristics used to quickly find invariant violations, 2. a strategy to shrink executions, and 3. integrating a testing approach with the overall Repliss tool.

CCS CONCEPTS

• **Information systems → Inconsistent data**; • **Software and its engineering → Consistency**; **Software testing and debugging**;

KEYWORDS

Property-based Testing, Weak Consistency

ACM Reference format:
Peter Zeller. 2017. Testing properties of weakly consistent programs with Repliss. In *Proceedings of PaPoC'17, Belgrade, Serbia, April 23, 2017,* 5 pages.
DOI: http://dx.doi.org/10.1145/3064889.3064893

1 BACKGROUND: REPLISS

Repliss is a tool for verifying and testing code interacting with weakly consistent databases. A Repliss program consists of a set of procedures and a definition of the database schema with relevant type definitions. Repliss procedures are supposed to provide high-level operations on the persistent data of the application to clients. They are implemented in a simple, imperative language which includes an `atomic`-statement to execute a block of operations in a database transaction. Procedures are called by clients and run concurrently, possibly on different servers and connected to different replicas of a shared database. For simplicity, we assume that procedures are only called by clients and not by other procedures of the program.

The database schema of a Repliss program is given by a set of query- and update-operations on CRDT-objects [14]. The result of queries is specified using first order logical formulas over the history of operations. This is very flexible and allows to use any CRDT, whose specification can be expressed in the logic.

Users can express properties about the behavior of a Repliss program using first order formulas. These formulas can use the queries defined on the datatypes to express data invariants, but they can also address the history of database operations and procedure invocations to specify temporal properties (as used by the userbase example in Section 2).

From the program and the specified properties, Repliss can derive verification conditions. We use Why3 [9] as the verification backend, so we can use automated theorem provers like Z3 [8] to discharge the verification conditions and thereby prove the program correct automatically. For complex problems, it is possible to export the verification conditions to Isabelle/HOL [13] and prove them manually. The latter can also give interesting insights into why an automated proof attempt failed, however it requires a lot of manual effort. Most often, users will first specify some desired properties, run Repliss and find that it fails to verify the property automatically. This is either, because there is a bug in the program as the user did not anticipate all possible concurrent interactions. However, it can also happen that additional invariants are required to complete the automated proof, or that the automated prover is not powerful enough.

To aid the programmer in localizing the issue, we extended Repliss with an automated testing tool. Testing can quickly reveal bugs in the program or specification by giving counter examples. However, it cannot find counterexamples for failed verification attempts of a correct program. Our tool works similar to tools like QuickCheck [5], which means that no additional inputs need to be given by the user and that the tool can automatically shrink counter examples to present them in a (locally) minimal form.

There are existing approaches to use QuickCheck for testing concurrent code. Pulse [6] is a custom scheduler, which explores different schedules at the level of Erlang processes. Our approach only considers the relevant nondeterminism which reduces the space of possible executions. Gambit [7] is another testing tool, which explores low-level schedules. It uses heuristics which are mainly based on partial order reduction, which do not apply to our setting, since we do not use linear executions.

For testing weakly consistent programs, there are surprisingly few tools. Netflix described using a tool named Chaos Monkey [3], a tool which injects network faults at runtime to test the robustness of the application. A more systematic approach is Chapar [12], which includes a bounded model checker for weakly consistent programs. In contrast to our work, it can enumerate all possible schedules, but it only checks small executions with given parameters, whereas we explore different parameters and executions automatically.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PaPoC'17, Belgrade, Serbia
© 2017 ACM. 978-1-4503-4933-8/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064889.3064893

2 EXAMPLES

In this section we use two examples to demonstrate, how our testing tool can be used to find bugs. Figure 1 shows an example program implementing a database of users. The function `registerUser` creates a new user account with the given data and returns the unique identifier of the newly created user. To update the mail address of a user with a given identifier, there is a function `updateMail`. To remove a user from the system, `removeUser` can be called. The data of a user can be retrieved via `getUser`, which returns a record with the name and the email address of a given user, or `not_found` when the user does not exist.

The database schema is given by defining the available operations and queries on the database and by choosing a semantics for the query operations (not shown here). The example employs a map data structure with operations `mapWrite` and `mapDelete` to store the users. The `mapWrite` operation takes two keys, the `UserId` and a field name (`f_name` or `f_mail`). Choosing a suitable CRDT semantics is essential for the correctness of the application. The userbase example only works correctly if `mapDelete` affects all prior and concurrent `mapWrite` operations, so a delete-wins map CRDT can be used.

For this application, we want to verify that calling `getUser(u)` after `removeUser(u)` for a user with unique identifier `u` returns `notFound`. We can specify this with the following invariant:

```
invariant forall r: invocationId, g: invocationId, u: UserId  ::
     r.info == removeUser(u)
  && g.info == getUser(u)
  && r happened before g
  ==> g.result == getUser_res(notFound())
```

This property can be verified by Repliss in one second[1]. The random test executor does not find a counter example and needs 2 seconds for executing 100 actions, 6 seconds for 200 action, 40 seconds for 400 actions and 240 seconds for 800 actions. In this example one invocation on average consists of roughly 4 actions. The numbers show that the executor does not scale well for long executions. This is mainly because of the straight forward evaluation strategy we are using at the moment. We can avoid this scalability problem, by running multiple shorter executions with different random seeds instead of running long executions.

When we introduce a bug by removing the atomic block in the `updateMail` procedure, our automatic testing tool finds the counter example shown in Figure 2. The graph is automatically generated for the Repliss web interface[2]. A failing example with 19 invocations is found after 600ms and reduced to the shown example with only 4 invocations in 2600ms. The outer boxes in the visualizations represent invocations. Each invocation can contain several boxes representing transactions and each transaction can again contain several calls to the database. The causal dependencies between database calls are denoted by arrows. As shown in Figure 2, the invariant can be violated, when a concurrent `mapDelete` call becomes visible before the second database call in `updateMail`.

When we reintroduce the atomic block and use an add-wins instead of a remove-wins CRDT, the testing tool also finds the bug and displays the counter example in Figure 3. In this case the problem is found after 300ms and 12 invocations and then reduced to 4 invocations in 2100ms. The example executions use "???" as a dummy result for calls to `mapGet`. We underspecified the `mapGet` query, since it is not important for this example, yet it is still possible to execute tests.

[1]All tests were executed on a laptop with 16Gb of RAM, i5-5500U processor, Ubuntu 16.04, Scala 2.11, Java OpenJDK-8, Why3-0.87.2, and Z3 4.5.1

[2]The Repliss web interface is available under https://softech.cs.uni-kl.de/repliss/

```
def registerUser(name: String, mail: String): UserId
  var u: UserId
  u = new UserId
  atomic
    call mapWrite(u, f_name(), name)
    call mapWrite(u, f_mail(), mail)
  return u

def updateMail(id: UserId, newMail: String)
  var uExists: boolean
  atomic
    uExists = mapExists(id)
    if (uExists)
      call mapWrite(id, f_mail(), newMail)

def removeUser(id: UserId)
  call mapDelete(id)

def getUser(id: UserId): getUserResult
  atomic
    if (mapExists(id))
      return found(mapGet(id, f_name()), mapGet(id, f_mail()))
    else
      return notFound()
```

Figure 1: Userbase Example: Procedures

Example 2: Set size. To challenge our testing tool, we ran it on the artificial example shown in Figure 4. The example consists of one procedure, which would ensure that at most one element is contained in the set if the code was executed in a sequential setting. We specified that the cardinality of the set is always less than 3. This means that a counterexample requires at least 3 parallel invocations with different elements and one action to merge the three states. Repliss was able to find an example with 6 invocations in 200ms and reduced it to the example in Figure 5 in 2000ms. This example also shows a limitation of our approach: A set size of n would require to have at least n different elements in the domain, however we only use a fixed number of values in tests (see Section 4).

3 SYSTEM MODEL

Repliss assumes that a database with transactional causal$^+$ consistency like Antidote [2] is used by the application. The formalization of the semantics is similar to formalizations used by Gotsman et al. [4, 11], but we describe it operationally, corresponding to our implementation. In particular, our semantics are deterministic for a given trace of actions and stable under removal of actions, which enables efficient shrinking of counter examples (see Section 5). The system state is modeled by the following components:

```
calls: Map[CallId, CallInfo]
transactions: Map[TransactionId, TransactionInfo]
invocations: Map[InvocationId, InvocationInfo]
knownIds: Map[IdType, Set[AnyValue]]
localStates: Map[InvocationId, LocalState]
```

All calls to the database (queries and update operations) are stored in `calls`. Each call has information about its operation, causal dependencies, originating transaction and origination procedure invocation. Information about transactions is stored in `transactions`.

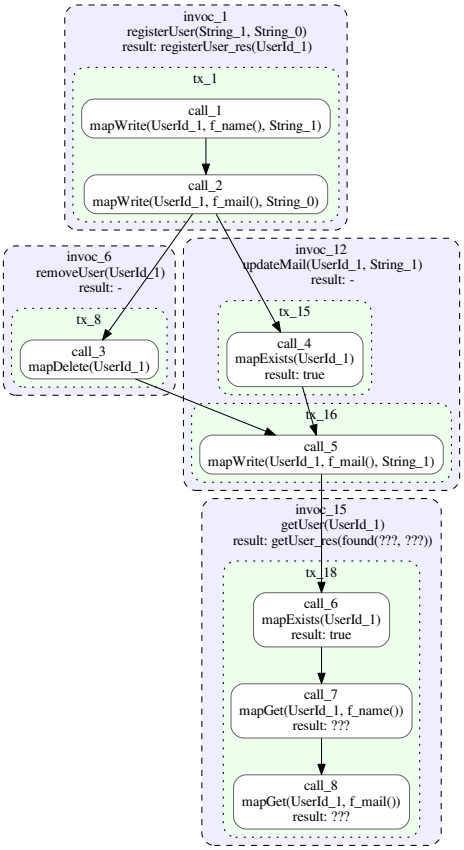


Figure 2: Counter example with missing transaction.

Each transaction stores information about the calls visible to the transactions (the transaction snapshot), the originating invocation, the calls executed inside the transaction and a flag stating whether the transaction was already committed. All invocations of procedures in the Repliss program are stored in `invocations`. For each invocation we store the called procedure and given arguments and for completed procedures also the returned value. Repliss contains a special class of types, which represent unique identifiers. These are called `IdType` and can only be generated by the system, so clients can only use a value of such a type after it has been returned by some previous procedure invocation. Therefore, we keep track of the identifiers known to clients in `knownIds`. Finally, each invocation has a local state stored in `localStates`. A local state consists of the following components:

```
varValues: Map[LocalVar, AnyValue]
todo: List[StatementOrAction]
waitingFor: LocalWaitingFor
currentTransaction: Option[TransactionInfo]
visibleCalls: Set[CallId]
```

The `varValues` map contains the current values of the local variables. The `todo` List represents a continuation consisting of the statements or actions which still have to be executed. When the execution of a procedure is suspended, the `waitingFor` field is describing the condition on which the execution is waiting. Additionally,

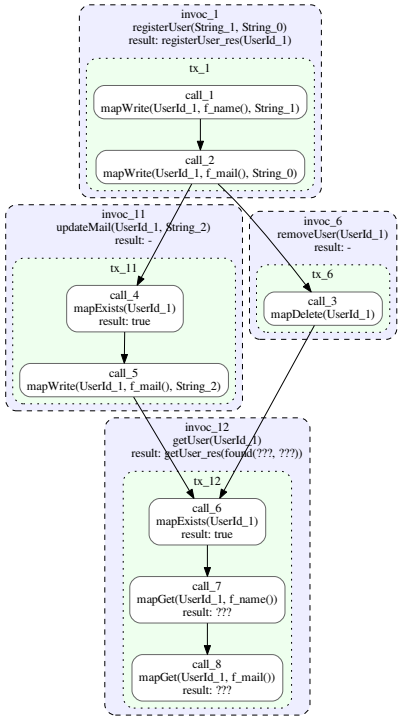


Figure 3: Counter example with wrong CRDT.

```
def store(e: Element) {
    atomic {
        if (elementSet_isEmpty()) {
            call elementSet_add(e)
        }
    }
}
```

Figure 4: Example with restricted set size

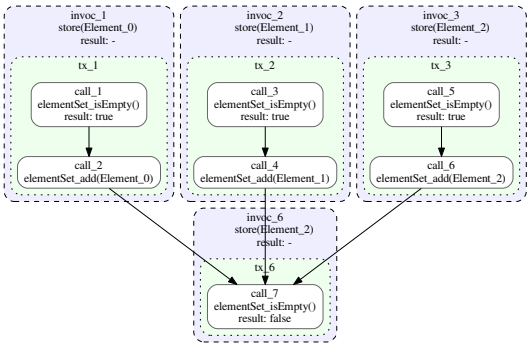


Figure 5: Counter example with 3 elements in the set.

the current transaction and the set of locally visible calls are stored in the local state.

An execution always starts in the initial state, where all maps are empty. This ensures, that we only reach database states that are reachable in practice.

To model concurrent executions, we allow suspending the execution of a local procedure at certain points, which could be observed by other concurrent invocations. We also suspend the execution at other points of nondeterministic decisions, so that these decisions can be delegated to the test executor. After reaching a suspended state on one invocation, the system can perform one of the following actions:

```
CallAction(procname: String, args: List[AnyValue])
StartTransaction(newTransactionId: TransactionId,
    pulledTransaction: Set[TransactionId])
NewId(id: Int)
Fail()
Return()
InvariantCheck()
```

In addition to the arguments shown here, each action also includes an invocation-identifier. A `CallAction` simply starts a new invocation of the given procedure with the given arguments and executes the procedure until it reaches the first suspension point. This is either a start of a new transaction (we transform the program, so that every database call is executed in a transaction), a statement to create a new unique identifier, or a return-statement. A transaction is started with the `StartTransaction` action, which gives the new transaction the `newTransactionId`. Transactions work on a snapshot, which is determined by the locally visible calls at the start of the transaction. When a transaction is started, the calls from the completed `pulledTransaction`s are pulled into the locally visible calls by adding them to the set of already visible calls. In addition, all causal dependencies of the newly pulled calls are added as well. This ensures, that the snapshot is causally consistent and adheres to the atomic visibility of transactions. Dependencies of database calls are recorded when a call is performed. Then all calls, which are locally visible, are added to the dependencies of the new call. This means, that our model tracks causality globally and not just per object.

The other actions are straightforward: `Fail` simulates a local crash by terminating an invocation without completing it. `Return` completes an invocation and stores the return-value. `InvariantCheck` checks the invariant and does not change the state.

3.1 Executing Specifications

One problem in executing the system model described above, is that it requires evaluating first order logical formulas for checking properties and for determining the results of database queries, which can also be specified by the user. As we only consider finite executions, we only have to evaluate formulas in finite models. Therefore, we can evaluate universal and existential quantifiers by checking them for all possible objects. This evaluation strategy is expensive, but by limiting the domain of types and by using short-circuiting evaluation, this approach is usable for the example we tested it on. Still, it is the major performance bottleneck in the current implementation, so future work will have to focus on improving the evaluation strategy. In the current implementation, we avoided the performance problems, by not checking the invariant in every step. Instead we introduced the `InvariantCheck` action, which is only executed occasionally.

4 HEURISTICS FOR FINDING BUGS

Our first (naive) approach of randomly exploring the space of concurrent executions failed to find invariant violations for our examples. We had to introduce heuristics to guide the search towards interesting cases of concurrency. These heuristics are a direct consequence of the following observations:

(1) Bugs happen more often, when there are few objects with many concurrent accesses.
(2) When a procedure consists of several transactions, bugs often appear when many changes are pulled in between transactions.
(3) Causal consistency tends to tame the chaos introduced by randomness, so a random choice must consider causality.
(4) Most bugs can be triggered by a short sequence of actions. Longer sequences of random actions can lead to save states, where no bugs can appear (for example, deleted objects and final state of a state-machine are often safe and cannot become unsafe again).

Point 1 was the easiest to address. We simply limit the size of the domain. For our examples a size of 3 values per primitive type worked well (e.g. we only use 3 different strings in executions). For id-types, we inspect the results of procedures and when a procedure has generated more than 3 unique identifiers, we stop generating new calls to the procedure.

Points 2-4 were not addressed well in our initial approach, where we simply selected a random subset of all transactions as the set of pulled transactions. Because all causal dependencies are included in a pull, picking a transaction with many dependencies, pulled in almost all transactions, which led to an almost linear history and did not reveal many bugs. Furthermore, it was unlikely that the next transaction in the same invocation would start on a substantially different snapshot. We addressed these issues with the following approach: First we calculate the set of transactions, which are not yet visible in the current invocation. Then we either pick one or two random transactions from this set, which simulates pulling in changes from one or two other replicas. This addresses point 2, since we always pull in new transactions when possible (an exception is the first pull in an invocation, where we allow starting from the empty state). To address point 3 and 4, we introduced a bias towards older transactions, so that we avoid including a big set of causal dependencies.

5 SHRINKING COUNTER EXAMPLES

When we find an invariant violation, we try to shrink the execution in order to present a small counter example to the user. An execution is defined by the trace of actions, which were randomly generated before. The trace includes all nondeterministic choices made by the system and therefore a trace can be executed deterministically. This property is important for replaying traces.

To allow efficient shrinking of the trace, the execution also has to be stable when removing some actions from the trace: this should only have minimal effects on other actions. We achieve this with the following two design decisions: First, we fix all generated identifiers

in the actions of the trace, so that identifiers are not affected by removing previous actions. Then, we ensure that actions can still be executed, even when the context has changed. For example, a `StartTransaction` action can include pulled transactions, which have already been removed and we simply ignore them. Moreover, the set of pulled transactions stored in the action already includes all causal dependencies, so removing one transaction has a minimal affect on the overall set of pulled transactions.

When we encounter an invalid action during execution, we simply ignore it and report the invalid action to the shrinking process. That way, we can directly remove all actions that have been invalidated by removing a single action. For example, when we remove the call to a procedure that returns a new unique identifier, this approach removes all actions in that call, as well as all calls using the generated identifier.

The shrinking algorithm itself is then straightforward: We try removing an action from the current trace, starting with the first action. When the reduced trace still triggers the invariant violation, we continue shrinking the reduced trace. Otherwise we try to remove the next action from the trace instead and continue as above.

The outcome of this process is visualized using the GraphViz [1, 10] tools `tred` (for removing transitive causality edges) and `dot` for the layout of the graph. Examples of the resulting visualizations are Figures 2, 3, and 5.

6 CONCLUSION

The addition of a testing tool to Repliss makes it easier to find valid invariants. Bugs in the code or in the specification can be discovered directly and fixed by studying the minimized counter example provided by the testing tool. In comparison to model checking approaches, random testing has the benefit of being easier to implement and maintain and it has shown to be useful for improving the usability of the Repliss tool.

In the future we plan to verify and test larger applications with Repliss. This will show, whether our testing approach scales to realistic examples. We expect that more complex examples will require a more efficient evaluation and may require some user interaction: Similar to QuickCheck, it might be necessary to provide custom generators if there are stronger preconditions on inputs.

Code availability

The code developed for Repliss is available at https://softech-git.cs.uni-kl.de/zeller/repliss. The tool can also be tried using the web interface at https://softech.cs.uni-kl.de/repliss/.

ACKNOWLEDGMENTS

I would like to thank Annette Bieniusa for useful feedback and discussions and the anonymous reviewers for their comments.

REFERENCES

[1] Graphviz - graph visualization software. http://www.graphviz.org/. Accessed: 2017-02-16.

[2] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *36th IEEE International Conference on Distributed Computing Systems, ICDCS*, 2016.

[3] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html, 2012. Accessed: 2017-03-12.

[4] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.

[5] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[6] Koen Claessen, Michal H. Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf T. Wiger. Finding race conditions in erlang with quickcheck and PULSE. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP*, 2009.

[7] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, 2010.

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*, 2008.

[9] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, 2013.

[10] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11), 2000.

[11] Alexey Gotsman and Hongseok Yang. Composite replicated data types. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP*, 2015.

[12] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, 2016.

[13] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[14] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.

Non-uniform Replication

Gonçalo Cabrita[1] and Nuno Preguiça[2]

1 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
g.cabrita@campus.fct.unl.pt
2 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
nuno.preguica@fct.unl.pt

Abstract

Replication is a key technique in the design of efficient and reliable distributed systems. As information grows, it becomes difficult or even impossible to store all information at every replica. A common approach to deal with this problem is to rely on partial replication, where each replica maintains only a part of the total system information. As a consequence, a remote replica might need to be contacted for computing the reply to some given query, which leads to high latency costs particularly in geo-replicated settings. In this work, we introduce the concept of non-uniform replication, where each replica stores only part of the information, but where all replicas store enough information to answer every query. We apply this concept to eventual consistency and conflict-free replicated data types. We show that this model can address useful problems and present two data types that solve such problems. Our evaluation shows that non-uniform replication is more efficient than traditional replication, using less storage space and network bandwidth.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Non-uniform Replication; Partial Replication; Replicated Data Types; Eventual Consistency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.0

1 Introduction

Many applications run on cloud infrastructures composed by multiple data centers, geographically distributed across the world. These applications usually store their data on geo-replicated data stores, with replicas of data being maintained in multiple data centers. Data management in geo-replicated settings is challenging, requiring designers to make a number of choices to better address the requirements of applications.

One well-known trade-off is between availability and data consistency. Some data stores provide strong consistency [5, 17], where the system gives the illusion that a single replica exists. This requires replicas to coordinate for executing operations, with impact on the latency and availability of these systems. Other data stores [7, 11] provide high-availability and low latency by allowing operations to execute locally in a single data center eschewing a linearizable consistency model. These systems receive and execute updates in a single replica before asynchronously propagating the updates to other replicas, thus providing very low latency.

With the increase of the number of data centers available to applications and the amount of information maintained by applications, another trade-off is between the simplicity of maintaining all data in all data centers and the cost of doing so. Besides sharding data among multiple machines in each data center, it is often interesting to keep only part of the data in each data center to reduce the costs associated with data storage and running


© Gonçalo Cabrita and Nuno Preguiça;
licensed under Creative Commons License CC-BY

protocols that involve a large number of replicas. In systems that adopt a partial replication model [22, 25, 6], as each replica only maintains part of the data, it can only locally process a subset of the database queries. Thus, when executing a query in a data center, it might be necessary to contact one or more remote data centers for computing the result of the query.

In this paper we explore an alternative partial replication model, the non-uniform replication model, where each replica maintains only part of the data but can process all queries. The key insight is that for some data objects, not all data is necessary for providing the result of read operations. For example, an object that keeps the top-K elements only needs to maintain those top-K elements in every replica. However, the remaining elements are necessary if a remove operation is available, as one of the elements not in the top needs to be promoted when a top element is removed.

A top-K object could be used for maintaining the leaderboard in an online game. In such system, while the information for each user only needs to be kept in the data center closest to the user (and in one or two more for fault tolerance), it is important to keep a replica of the leaderboard in every data center for low latency and availability. Currently, for supporting such a feature, several designs could be adopted. First, the system could maintain an object with the results of all players in all replicas. While simple, this approach turns out to be needlessly expensive in both storage space and network bandwidth when compared to our proposed model. Second, the system could move all data to a single data center and execute the computation in that data center or use a data processing system that can execute computations over geo-partitioned data [10]. The result would then have to be sent to all data centers. This approach is much more complex than our proposal, and while it might be interesting when complex machine learning computations are executed, it seems to be an overkill in a number of situations.

We apply the non-uniform replication model to eventual consistency and Conflict-free Replicated Data Types [23], formalizing the model for an operation-based replication approach. We present two useful data type designs that implement such model. Our evaluation shows that the non-uniform replication model leads to high gains in both storage space and network bandwidth used for synchronization when compared with state-of-the-art replication based alternatives.

In summary, this paper makes the following contributions:

- The proposal of the non-uniform replication model, where each replica only keeps part of the data but enough data to reply to every query;
- The definition of non-uniform eventual consistency (NuEC), the identification of sufficient conditions for providing NuEC and a protocol that enforces such conditions relying on operation-based synchronization;
- Two useful replicated data type designs that adopt the non-uniform replication model (and can be generalized to use different filter functions);
- An evaluation of the proposed model, showing its gains in term of storage space and network bandwidth.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 describes the non-uniform replication model. Section 4 applies the model to an eventual consistent system. Section 5 introduces two useful data type designs that follow the model. Section 6 compares our proposed data types against state-of-the-art CRDTs.

2 Related Work

Replication: A large number of replication protocols have been proposed in the last decades [8, 27, 15, 16, 2, 21, 17]. Regarding the contents of the replicas, these protocols can be divided in those providing full replication, where each replica maintains the full database state, and partial replication, where each replica maintains only a subset of the database state.

Full replication strategies allow operations to concurrently modify all replicas of a system and, assuming that replicas are mutually consistent, improves availability since clients may query any replica in the system and obtain an immediate response. While this improves the performance of read operations, update operations now negatively affect the performance of the system since they must modify every replica which severely affects middle-scale to large-scale systems in geo-distributed settings. This model also has the disadvantage of limiting the system's total capacity to the capacity of the node with fewest resources.

Partial replication [3, 22, 25, 6] addresses the shortcomings of full replication by having each replica store only part of the data (which continues being replicated in more than one node). This improves the scalability of the system but since each replica maintains only a part of the data, it can only locally process a subset of queries. This adds complexity to the query processing, with some queries requiring contacting multiple replicas to compute their result. In our work we address these limitations by proposing a model where each replica maintains only part of the data but can reply to any query.

Despite of adopting full or partial replication, replication protocols enforce strong consistency [17, 5, 18], weak consistency [27, 7, 15, 16, 2] or a mix of these consistency models [24, 14]. In this paper we show how to combine non-uniform replication with eventual consistency. An important aspect in systems that adopt eventual consistency is how the system handles concurrent operations. CRDTs have been proposed as a technique for addressing such challenge.

CRDTs: Conflict-free Replicated Data Types [23] are data types designed to be replicated at multiple replicas without requiring coordination for executing operations. CRDTs encode merge policies used to guarantee that all replicas converge to the same value after all updates are propagated to every replica. This allows an operation to execute immediately on any replica, with replicas synchronizing asynchronously. Thus, a system that uses CRDTs can provide low latency and high availability, despite faults and network latency. With these guarantees, CRDTs are a key building block for providing eventual consistency with well defined semantics, making it easier for programmers to reason about the system evolution.

When considering the synchronization process, two main types of CRDTs have been proposed: state-based CRDT, where replicas synchronize pairwise, by periodically exchanging the state of the replicas; and operation-based CRDTs, where all operations need to be propagated to all replicas.

Delta-based CRDTs [1] improve upon state-based CRDTs by reducing the dissemination cost of updates, sending only a delta of the modified state. This is achieved by using *delta-mutators*, which are functions that encode a delta of the state. Linde et. al [26] propose an improvement to delta-based CRDTs that further reduce the data that need to be propagated when a replica first synchronizes with some other replica. This is particularly interesting in peer-to-peer settings, where the synchronization partners of each replica change frequently. Although delta-based CRDTs reduce the network bandwidth used for synchronization, they continue to maintain a full replication strategy where the state of quiescent replicas is equivalent.

Computational CRDTs [19] are an extension of state-based CRDTs where the state of the object is the result of a computation (e.g. the average, the top-K elements) over the executed updates. As with the model we propose in this paper, replicas do not need to have equivalent states. The work we present in this paper extends the initial ideas proposed in computational CRDTs in several aspects, including the definition of the non-uniform replication model, its application to operation-based eventual consistency and the new data type designs.

3 Non-uniform replication

We consider an asynchronous distributed system composed by n nodes. Without loss of generality, we assume that the system replicates a single object. The object has an interface composed by a set of read-only operations, $\mathcal{Q}$, and a set of update operations, $\mathcal{U}$. Let $\mathcal{S}$ be the set of all possible object states, the state that results from executing operation o in state $s \in \mathcal{S}$ is denoted as $s \bullet o$. For a read-only operation, $q \in \mathcal{Q}$, $s \bullet q = s$. The result of operation $o \in \mathcal{Q} \cup \mathcal{U}$ in state $s \in \mathcal{S}$ is denoted as $o(s)$ (we assume that an update operation, besides modifying the state, can also return some result).

We denote the state of the replicated system as a tuple $(s_1, s_2, \ldots, s_n)$, with s_i the state of the replica i. The state of the replicas is synchronized by a replication protocol that exchanges messages among the nodes of the system and updates the state of the replicas. For now, we do not consider any specific replication protocol or strategy, as our proposal can be applied to different replication strategies.

We say a system is in a quiescent state for a given set of executed operations if the replication protocol has propagated all messages necessary to synchronize all replicas, i.e., additional messages sent by the replication protocol will not modify the state of the replicas. In general, replication protocols try to achieve a convergence property, in which the state of any two replicas is equivalent in a quiescent state.

▶ **Definition 1** (Equivalent state)**.** Two states, s_i and s_j, are *equivalent*, $s_i \equiv s_j$, iff the results of the execution of any sequence of operations in both states are equal, i.e., $\forall o_1, \ldots, o_n \in \mathcal{Q} \cup \mathcal{U}, o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$.

This property is enforced by most replication protocols, independently of whether they provide strong or weak consistency [13, 15, 27]. We note that this property does not require that the internal state of the replicas is the same, but only that the replicas always return the same results for any executed sequence of operations.

In this work, we propose to relax this property by requiring only that the execution of read-only operations return the same value. We name this property as *observable equivalence* and define it formally as follows.

▶ **Definition 2** (Observable equivalent state)**.** Two states, s_i and s_j, are *observable equivalent*, $s_i \overset{\circ}{\equiv} s_j$, iff the result of executing every read-only operation in both states is equal, i.e., $\forall o \in \mathcal{Q}, o(s_i) = o(s_j)$.

As read-only operations do not affect the state of a replica, the results of the execution of any sequence of read-only operations in two observable equivalent states will also be the same. We now define a non-uniform replication system as one that guarantees only that replicas converge to an observable equivalent state.

▶ **Definition 3** (Non-uniform replicated system)**.** We say that a replicated system is non-uniform if the replication protocol guarantees that in a quiescent state, the state of any two replicas is observable equivalent, i.e., in the quiescent state $(s_1, \ldots, s_n)$, we have $s_i \overset{\circ}{\equiv} s_j, \forall s_i, s_j \in \{s_1, \ldots, s_n\}$.

3.1 Example

We now give an example that shows the benefit of non-uniform replication. Consider an object *top-1* with three operations: (i) $add(name, value)$, an update operation that adds the pair to the top; (ii) $rmv(name)$, an update operation that removes all previously added pairs for *name*; (iii) $get()$, a query that returns the pair with the largest value (when more than one pair has the same largest value, the one with the smallest lexicographic name is returned).

Consider that $add(a, 100)$ is executed in a replica and replicated to all replicas. Later $add(b, 110)$ is executed and replicated. At this moment, all replicas know both pairs.

If later $add(c, 105)$ executes in some replica, the replication protocol does not need to propagate the update to the other replicas in a non-uniform replicated system. In this case, all replicas are observable equivalent, as a query executed at any replica returns the same correct value. This can have an important impact not only in the size of object replicas, as each replica will store only part of the data, but also in the bandwidth used by the replication protocol, as not all updates need to be propagated to all replicas.

We note that the states that result from the previous execution are not equivalent because after executing $rmv(b)$, the get operation will return $(c, 105)$ in the replica that has received the $add(c, 105)$ we operation and $(b, 100)$ in the other replicas.

Our definition only forces the states to be observable equivalent after the replication protocol becomes quiescent. Different protocols can be devised giving different guarantees. For example, for providing linearizability, the protocol should guarantee that all replicas return $(c, 105)$ after the remove. This can be achieved, for example, by replicating the now relevant $(c, 105)$ update in the process of executing the remove.

In the remainder of this paper, we study how to apply the concept of non-uniform replication in the context of eventually consistent systems. The study of its application to systems that provide strong consistency is left for future work.

4 Non-uniform eventual consistency

We now apply the concept of non-uniform replication to replicated systems providing eventual consistency.

4.1 System model

We consider an asynchronous distributed system composed by n nodes, where nodes may exhibit fail-stop faults but not byzantine faults. We assume a communication system with a single communication primitive, $mcast(m)$, that can be used by a process to send a message to every other process in the system with reliable broadcast semantics. A message sent by a correct process is eventually received by all correct processes. A message sent by a faulty process is either received by all correct processes or none. Several communication systems provide such properties – e.g. systems that propagate messages reliably using anti-entropy protocols [8, 9].

An object is defined as a tuple $(\mathcal{S}, s^0, \mathcal{Q}, \mathcal{U}_p, \mathcal{U}_e)$, where $\mathcal{S}$ is the set of valid states of the object, $s^0 \in \mathcal{S}$ is the initial state of the object, $\mathcal{Q}$ is the set of read-only operations (or *queries*), $\mathcal{U}_p$ is the set of prepare-update operations and $\mathcal{U}_e$ is the set of effect-update operations.

A query executes only at the replica where the operation is invoked, its source, and it has no side-effects, i.e., the state of an object remains unchanged after executing the operation.

When an application wants to update the state of the object, it issues a prepare-update operation, $u_p \in \mathcal{U}_p$. A u_p operation executes only at the source, has no side-effects and generates an effect-update operation, $u_e \in \mathcal{U}_e$. At source, u_e executes immediately after u_p.

As only effect-update operations may change the state of the object, for reasoning about the evolution of replicas we can restrict our analysis to these operations. To be precise, the execution of a prepare-update operation generates an instance of an effect-update operation. For simplicity, we refer the instances of operations simply as operations. With O_i the set of operations generated at node i, the set of operations generated in an execution, or simply the set of operations in an execution, is $O = O_1 \cup \ldots \cup O_n$.

4.2 Non-uniform eventual consistency

For any given execution, with O the operations of the execution, we say a replicated system provides *eventual consistency* iff in a quiescent state: (i) every replica executed all operations of O; and (ii) the state of any pair of replicas is equivalent.

A sufficient condition for achieving the first property is to propagate all generated operations using reliable broadcast (and execute any received operation). A sufficient condition for achieving the second property is to have only commutative operations. Thus, if all operations commute with each other, the execution of any serialization of O in the initial state of the object leads to an equivalent state.

From now on, unless stated otherwise, we assume that all operations commute. In this case, as all serializations of O are equivalent, we denote the execution of a serialization of O in state s simply as $s \bullet O$.

For any given execution, with O the operations of the execution, we say a replicated system provides *non-uniform eventual consistency* iff in a quiescent state the state of any replica is observable equivalent to the state obtained by executing some serialization of O. As a consequence, the state of any pair of replicas is also observable equivalent.

For a given set of operations in an execution O, we say that $O_{core} \subseteq O$ is a set of core operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet O_{core}$. We define the set of operations that are irrelevant to the final state of the replicas as follows: $O_{masked} \subseteq O$ is a set of masked operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

▶ **Theorem 4** (Sufficient conditions for NuEC). *A replication system provides* non-uniform eventual consistency (NuEC) *if, for a given set of operations O, the following conditions hold: (i) every replica executes a set of core operations of O; and (ii) all operations commute.*

Proof. From the definition of core operations of O, and by the fact that all operations commute, it follows immediately that if a replica executes a set of core operations, then the final state of the replica is observable equivalent to the state obtained by executing a serialization of O. Additionally, any replica reaches an observable equivalent state. ◀

4.3 Protocol for non-uniform eventual consistency

We now build on the sufficient conditions for providing *non-uniform eventual consistency* to devise a correct replication protocol that tries to minimize the operations propagated to other replicas. The key idea is to avoid propagating operations that are part of a masked set. The challenge is to achieve this by using only local information, which includes only a subset of the executed operations.

Algorithm 1 presents the pseudo-code of an algorithm for achieving *non-uniform eventual consistency* – the algorithm does not address the durability of operations, which will be discussed later.

Algorithm 1 Replication algorithm for non-uniform eventual consistency

```
1:  S : state: initial s^0                                          ▷ Object state
2:  log_recv : set of operations: initial {}
3:  log_local : set of operations: initial {}                       ▷ Local operations not propagated
4:
5:  execOp(op): void                                                ▷ New operation generated locally
6:    log_local = log_local ∪ {op}
7:    S = S • op
8:
9:  opsToPropagate(): set of operations ▷ Computes the local operations that need to be propagated
10:   ops = maskedForever(log_local, S, log_recv)
11:   log_local = log_local \ ops
12:   opsImpact = hasObservableImpact(log_local, S, log_recv)
13:   opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
14:   return opsImpact ∪ opsPotImpact
15:
16: sync(): void                                                    ▷ Propagates local operations to remote replicas
17:   ops = opsToPropagate()
18:   compactedOps = compact(ops)                                   ▷ Compacts the set of operations
19:   mcast(compactedOps)
20:   log_coreLocal = {}
21:   log_local = log_local \ ops
22:   log_recv = log_recv ∪ ops
23:
24: on receive(ops): void                                           ▷ Process remote operations
25:   log_recv = log_recv ∪ ops
26:   S = S • ops
```

The algorithm maintains the state of the object and two sets of operations: log_{local}, the set of effect-update operations generated in the local replica and not yet propagated to other replicas; log_{recv}, the set of effect-update operations propagated to all replicas (including operations generated locally and remotely).

When an effect-update operation is generated, the *execOp* function is called. This function adds the new operation to the log of local operations and updates the local object state.

The function *sync* is called to propagate local operations to remote replicas. It starts by computing which new operations need to be propagated, compacts the resulting set of operations for efficiency purposes, multicasts the compacted set of operations, and finally updates the local sets of operations. When a replica receives a set of operations (line 24), the set of operations propagated to all nodes and the local object state are updated accordingly.

Function *opsToPropagate* addresses the key challenge of deciding which operations need to be propagated to other replicas. To this end, we divide the operations in four groups.

First, the *forever masked* operations, which are operations that will remain in the set of masked operations independently of the operations that might be executed in the future. In the top example, an operation that adds a pair masks forever all known operations that added a pair for the same element with a lower value. These operations are removed from the set of local operations.

Second, the *core* operations (*opsImpact*, line 12), as computed locally. These operations need to be propagated, as they will (typically) impact the observable state at every replica.

Third, the operations that might impact the observable state when considered in combination with other non-core operations that might have been executed in other replicas (*opsPotImpact*, line 13). As there is no way to know which non-core operations have been executed in other replicas, it is necessary to propagate these operations also. For example, consider a modified top object where the value associated with each element is the sum of the values of the pairs added to the object. In this case, an add operation that would not move an element to the top in a replica would be in this category because it could influence

the top when combined with other concurrent adds for the same element.

Fourth, the remaining operations that might impact the observable state in the future, depending on the evolution of the observable state. These operations remain in log_{local}. In the original top example, an operation that adds a pair that will not be in the top, as computed locally, is in this category as it might become the top element after removing the elements with larger values.

For proving that the algorithm can be used to provide non-uniform eventual consistency, we need to prove the following property.

▶ **Theorem 5.** *Algorithm 1 guarantees that in a quiescent state, considering all operations O in an execution, all replicas have received all operations in a core set O_{core}.*

Proof. To prove this property, we need to prove that there exists no operation that has not been propagated by some replica and that is required for any O_{core} set. Operations in the first category have been identified as masked operations independently of any other operations that might have been or will be executed. Thus, by definition of masked operations, a O_{core} set will not (need to) include these operations. The fourth category includes operations that do not influence the observable state when considering all executed operations – if they might have impact, they would be in the third category. Thus, these operations do not need to be in a O_{core} set. All other operations are propagated to all replicas. Thus, in a quiescent state, every replica has received all operations that impact the observable state. ◀

4.4 Fault-tolerance

Non-uniform replication aims at reducing the cost of communication and the size of replicas, by avoiding propagating operations that do not influence the observable state of the object. This raises the question of the durability of operations that are not immediately propagated to all replicas.

One way to solve this problem is to have the source replica propagating every local operation to f more replicas to tolerate f faults. This ensures that an operation survives even in the case of f faults. We note that it would be necessary to adapt the proposed algorithm, so that in the case a replica receives an operation for durability reasons, it would propagate the operation to other replicas if the source replica fails. This can be achieved by considering it as any local operation (and introducing a mechanism to filter duplicate reception of operations).

4.5 Causal consistency

Causal consistency is a popular consistency model for replicated systems [15, 2, 16], in which a replica only executes an operation after executing all operations that causally precede it [12]. In the non-uniform replication model, it is impossible to strictly adhere to this definition because some operations are not propagated (immediately), which would prevent all later operations from executing.

An alternative would be to restrict the dependencies to the execution of core operations. The problem with this is that the status of an operation may change by the execution of another operation. When a non-core operation becomes core, a number of dependencies that should have been enforced might have been missed in some replicas.

We argue that the main interest of causal consistency, when compared with eventual consistency, lies in the semantics provided by the object. Thus, in the designs that we present in the next section, we aim to guarantee that in a quiescent state, the state of the replicated objects provide equivalent semantics to that of a system that enforces causal consistency.

5 Non-uniform operation-based CRDTs

CRDTs [23] are data-types that can be replicated, modified concurrently without coordination and guarantee the eventual consistency of replicas given that all updates propagate to all replicas. We now present the design of two useful operation-based CRDTs [23] that adopt the non-uniform replication model. Unlike most operation-based CRDT designs, we do not assume that the system propagates operations in a causal order. These designs were inspired by the state-based computational CRDTs proposed by Navalho *et al.* [19], which also allow replicas to diverge in their quiescent state.

5.1 Top-K with removals NuCRDT

In this section we present the design of a non-uniform top-K CRDT, as the one introduced in section 3.1. The data type allows access to the top-K elements added and can be used, for example, for maintaining the leaderboard in online games. The proposed design could be adapted to define any CRDT that filters elements based on a deterministic function by replacing the *topK* function used in the algorithm by another filter function.

For defining the semantics of our data type, we start by defining the happens-before relation among operations. To this end, we start by considering the happens-before relation established among the events in the execution of the replicated system [12]. The events that are considered relevant are: the generation of an operation at the source replica, and the dispatch and reception of a message with a new operation or information that no new message exists. We say that operation op_i happens before operation op_j iff the generation of op_i happened before the generation of op_j in the partial order of events.

The semantics of the operations defined in the top-K CRDT is the following. The *add(el,val)* operation adds a new pair to the object. The *rmv(el)* operation removes any pair of *el* that was added by an operation that happened-before the *rmv* (note that this includes non-core add operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [23], where a remove has no impact on concurrent adds. The *get()* operation returns the top-K pairs in the object, as defined by the function *topK* used in the algorithm.

Algorithm 2 presents a design that implements this semantics. The prepare-update *add* operation generates an effect-update *add* that has an additional parameter consisting in a timestamp $(replicaid, val)$, with *val* a monotonically increasing integer. The prepare-update *rmv* operation generates an effect-update *rmv* that includes an additional parameter consisting in a vector clock that summarizes add operations that happened before the remove operation. To this end, the object maintains a vector clock that is updated when a new add is generated or executed locally. Additionally, this vector clock should be updated whenever a replica receives a message from a remote replica (to summarize also the adds known in the sender that have not been propagated to this replica).

Besides this vector clock, *vc*, each object replica maintains: (i) a set, *elems*, with the elements added by all *add* operations known locally (and that have not been removed yet); and (ii) a map, *removes*, that maps each element *id* to a vector clock with a summary of the add operations that happened before all removes of *id* (for simplifying the presentation of the algorithm, we assume that a key absent from the map has associated a default vector clock consisting of zeros for every replica).

The execution of an *add* consists in adding the element to the set of *elems* if the add has not happened before a previously received remove for the same element – this can happen as operations are not necessarily propagated in causal order. The execution of a *rmv* consists

Algorithm 2 Top-K NuCRDT with removals

```
1:  elems : set of ⟨id, score, ts⟩ : initial {}
2:  removes : map id ↦ vectorClock: initial []
3:  vc : vectorClock: initial []
4:
5:  GET() : set
6:    return {⟨id, score⟩ : ⟨id, score, ts⟩ ∈ topK(elems)}
7:
8:  prepare ADD(id, score)
9:    generate add(id, score, ⟨getReplicaId(), ++ vc[getReplicaId()]⟩)
10:
11: effect ADD(id, score, ts)
12:   if removes[id][ts.siteId] < ts.val then
13:     elems = elems ∪ {⟨id, score, ts⟩}
14:     vc[ts.siteId] = max(vc[ts.siteId], ts.val)
15:
16: prepare RMV(id)
17:   generate rmv(id,vc)
18:
19: effect RMV(id, vc_rmv)
20:   removes[id] = pointwiseMax(removes[id], vc_rmv)
21:   elems = elems \ {⟨id_0, score, ts⟩ ∈ elem : id = id_0 ∧ ts.val ≤ vc_rmv[ts.siteId]}
22:
23: maskedForever(log_local, S, log_recv): set of operations
24:   adds = {add(id_1, score_1, ts_1) ∈ log_local :
25:     (∃add(id_2, score_2, ts_2) ∈ log_local : id_1 = id_2 ∧ score_1 < score_2 ∧ ts_1.val < ts_2.val)∨
26:       (∃rmv(id_3, vc_rmv) ∈ (log_recv ∪ log_local) : id_1 = id_3 ∧ ts_1.val ≤ vc_rmv[ts_1.siteId]}
27:   rmvs = {rmv(id_1, vc_1) ∈ log_local : ∃rmv(id_2, vc_2) ∈ (log_local ∪ log_recv) : id_1 = id_2 ∧ vc_1 < vc_2}
28:   return adds ∪ rmvs
29:
30: mayHaveObservableImpact(log_local, S, log_recv): set of operations
31:   return {}                          ▷ This case never happens for this data type
32:
33: hasObservableImpact(log_local, S, log_recv): set of operations
34:   adds = {add(id_1, score_1, ts_1) ∈ log_local : ⟨id_1, score_1, ts_1⟩ ∈ topK(S.elems)}
35:   rmvs = {rmv(id_1, vc_1) ∈ log_local : (∃add(id_2, score_2, ts_2) ∈ (log_local ∪ log_recv) :
36:       ⟨id_2, score_2, ts_2⟩ ∈ topK(S.elems ∪ {⟨id_2, score_2, ts_2⟩}) ∧ id_1 = id_2 ∧ ts_2.val ≤ vc_1[ts_2.siteId])}
37:   return adds ∪ rmvs
38:
39: compact(ops): set of operations
40:   return ops                         ▷ This data type does not require compaction
```

in updating *removes* and deleting from *elems* the information for adds of the element that happened before the remove. To verify if an add has happened before a remove, we check if the timestamp associated with the add is reflected in the remove vector clock of the element (lines 12 and 21). This ensures the intended semantics for the CRDT, assuming that the functions used by the protocol are correct.

We now analyze the code of these functions.

Function maskedForever computes: the local adds that become masked by other local adds (those for the same element with a lower value) and removes (those for the same element that happened before the remove); the local removes that become masked by other removes (those for the same element that have a smaller vector clock). In the latter case, it is immediate that a remove with a smaller vector clock becomes irrelevant after executing the one with a larger vector clock. In the former case, a local add for an element is masked by a more recent local add for the same element but with a larger value as it is not possible to remove only the effects of the later add without removing the effect of the older one. A local add also becomes permanently masked by a local or remote remove that happened after the add.

Function mayHaveObservableImpact returns the empty set, as for having impact on

any observable state, an operation also has to have impact on the local observable state by itself.

Function hasObservableImpact computes the local operations that are relevant for computing the top-K. An add is relevant if the added value is in the top; a remove is relevant if it removes an add that would be otherwise in the top.

5.2 Top Sum NuCRDT

We now present the design of a non-uniform CRDT, Top Sum, that maintains the top-K elements added to the object, where the value of each element is the sum of the values added for the element. This data type can be used for maintaining a leaderboard in an online game where every time a player completes some challenge it is awarded some number of points, with the current score of the player being the sum of all points awarded. It could also be used for maintaining a top of the best selling products in an (online) store (or the top customers, etc).

The semantics of the operations defined in the Top Sum object is the following. The *add(id, n)* update operation increments the value associated with *id* by *n*. The *get()* read-only operation returns the top-K mappings, $id \rightarrow value$, as defined by the *topK* function (similar to the Top-K NuCRDT).

This design is challenging, as it is hard to know which operations may have impact in the observable state. For example, consider a scenario with two replicas, where the value of the last element in the top is 100. If the known score of an element is 90, an add of 5 received in one replica may have impact in the observable state if the other replica has also received an add of 5 or more. One approach would be to propagate these operations, but this would lead to propagating all operations.

To try to minimize the number of operations propagated we use the following heuristic inspired by the demarcation protocol and escrow transactions [4, 20]. For each id that does not belong to the top, we compute the difference between the smallest value in the top and the value of the id computed by operations known in every replica – this is how much must be added to the id to make it to the top: let d be this value. If the sum of local adds for the id does not exceed $\frac{d}{num.replicas}$ in any replica, the value of id when considering adds executed in all replicas is smaller than the smallest element in the top. Thus, it is not necessary to propagate add operations in this case, as they will not affect the top.

Algorithm 3 presents a design that implements this approach. The state of the object is a single variable, *state*, that maps identifiers to their current values. The only prepare-update operation, *add*, generates an effect-update *add* with the same parameters. The execution of an effect-update *add(id, n)* simply increments the value of *id* by *n*.

Function maskedForever returns the empty set, as operations in this design can never be forever masked.

Function mayHaveObservableImpact computes the set of *add* operations that can potentially have an impact on the observable state, using the approach previously explained.

Function hasObservableImpact computes the set of *add* operations that have their corresponding *id* present in the top-K. This guarantees that the values of the elements in the top are kept up-to-date, reflecting all executed operations.

Function compact takes a set of *add* operations and compacts the *add* operations that affect the same identifier into a single operation. This reduces the size of the messages sent through the network and is similar to the optimization obtained in delta-based CRDTs [1].

Algorithm 3 Top Sum NuCRDT

```
1: state : map id ↦ sum: initial []
2:
3: GET() : map
4:   return topK(state)
5:
6: prepare ADD(id, n)
7:   generate add(id, n)
8:
9: effect ADD(id, n)
10:  state[id] = state[id] + n
11:
12: MASKEDFOREVER(log_local, S, log_recv): set of operations
13:  return {}                          ▷ This case never happens for this data type
14:
15: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
16:  top = topK(S.state)
17:  adds = {add(id, _) ∈ log_local : s = sum_val({add(i, n) ∈ log_local : i = id})
18:    ∧ s ≥ ((min(sum(top)) − (S.state[id] − s)) / getNumReplicas())}
19:  return adds
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
22:  top = topK(S.state)
23:  adds = {add(id, _) ∈ log_local : id ∈ ids(top)}
24:  return adds
25:
26: COMPACT(ops): set of operations
27:  adds = {add(id, n) : id ∈ {i : add(i, _) ∈ ops}∧ n = sum({k : add(id_1, k) ∈ ops : id_1 = id})}
28:  return adds
```

5.3 Discussion

The goal of non-uniform replication is to allow replicas to store less data and use less bandwidth for replica synchronization. Although it is clear that non-uniform replication cannot be useful for all data, we believe that the number of use cases is large enough for making non-uniform replication interesting in practice. We now discuss two classes of data types that can benefit from the adoption of non-uniform replication.

The first class is that of data types for which the result of queries include only a subset of the data in the object. In this case two different situations may occur: (i) it is possible to compute locally, without additional information, if some operation is relevant (and needs to be propagated to all replicas); (ii) it is necessary to have additional information to be able to decide if some operation is relevant. The Top-K CRDT presented in section 5.1 is an example of the former. Another example includes a data type that returns a subset of the elements added based on a (modifiable) user-defined filter – e.g. in a set of books, the filter could select the books of a given genre, language, etc. The Top-Sum CRDT presented in section 5.2 is an example of the latter. Another example includes a data type that returns the 50^{th} percentile (or others) for the elements added – in this case, it is only necessary to replicate the elements in a range close to the 50^{th} percentile and replicate statistics of the elements smaller and larger than the range of replicated elements.

In all these examples, the effects of an operation that in a given moment do not influence the result of the available queries may become relevant after other operations are executed – in the Top-K with removes due to a remove of an element in the top; in the filtered set due to a change in the filter; in the Top-Sum due to a new add that makes an element relevant; and in the percentile due to the insertion of elements that make the 50^{th} percentile change. We note that if the relevance of an operation cannot change over time, the non-uniform CRDT would be similar to an optimized CRDT that discard operations that are not relevant before propagating them to other replicas.

A second class is that of data types with queries that return the result of an aggregation over the data added to the object. An example of this second class is the Histogram CRDT presented in the appendix. This data type only needs to keep a count for each element. A possible use of this data type would be for maintaining the summary of classifications given by users in an online shop. Similar approaches could be implemented for data types that return the result of other aggregation functions that can be incrementally computed [19].

A data type that supports, besides adding some information, an operation for removing that information would be more complex to implement. For example, in an Histogram CRDT that supports removing a previously added element, it would be necessary that concurrently removing the same element would not result in an incorrect aggregation result. Implementing such CRDT would require detecting and fixing these cases.

6 Evaluation

In this section we evaluate our data types that follow the non-uniform replication model. To this end, we compare our designs against state-of-the-art CRDT alternatives: delta-based CRDTs [1] that maintain full object replicas efficiently by propagating updates as deltas of the state; and computational CRDTs [19] that maintain non-uniform replicas using a state-based approach.

Our evaluation is performed by simulation, using a discrete event simulator. To show the benefit in terms of bandwidth and storage, we measure the total size of messages sent between replicas for synchronization (total payload) and the average size of replicas.

We simulate a system with 5 replicas for each object. Both our designs and the computational CRDTs support up to 2 replica faults by propagating all operations to, at least, 2 other replicas besides the source replica. We note that this limits the improvement that our approach could achieve, as it is only possible to avoid sending an operation to two of the five replicas. By either increasing the number of replicas or reducing the fault tolerance level, we could expect that our approach would perform comparatively better than the delta-based CRDTs.

6.1 Top-K with removals

We begin by comparing our Top-K design (*NuCRDT*) with a delta-based CRDT set [1] (*Delta CRDT*) and the top-K state-based computational CRDT design [19] (*CCRDT*).

The top-K was configured with K equal to 100. In each run, 500000 update operations were generated for 10000 Ids and with scores up to 250000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Given the expected usage of top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Figures 1 and 2 show the results for workloads with 5% and 0.05% of removes respectively (the other operations are adds).

In both workloads our design achieves a significantly lower bandwidth cost when compared to the alternatives. The reason for this is that our design only propagates operations that will be part of the top-K. In the delta-based CRDT, each replica propagates all new updates and not only those that are part of the top. In the computational CRDT design, every time the top is modified, the new top is propagated. Additionally, the proposed design of computational CRDTs always propagates removes.

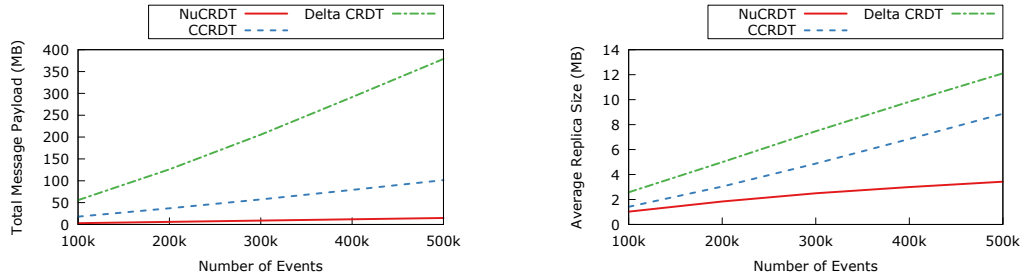


Figure 1 Top-K with removals: payload size and replica size, workload of 95/5

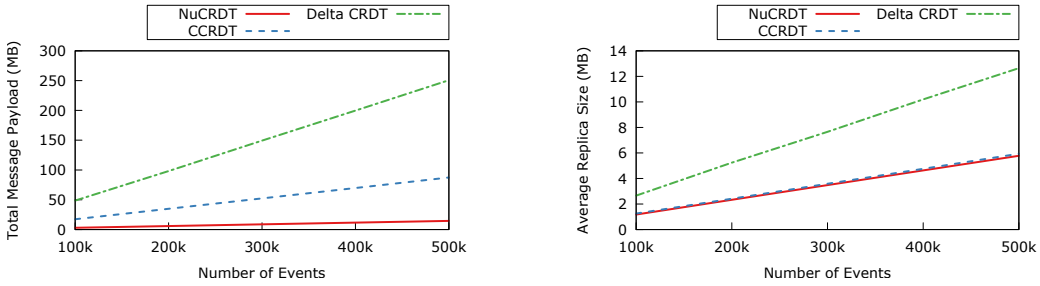


Figure 2 Top-K with removals: payload size and replica size, workload of 99.95/0.05

The results for the replica size show that our design is also more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information from remote operations received for guaranteeing fault-tolerance and those that have influenced the top-K at some moment in the execution. The computational CRDT design additionally keeps information about all removes. The delta-based CRDT keeps information about all elements that have not been removed or overwritten by a larger value. We note that as the percentage of removes approaches zero, the replica sizes of our design and that of computational CRDT starts to converge to the same value. The reason for this is that the information maintained in both designs is similar and our more efficient handling of removes starts becoming irrelevant. The opposite is also true: as the number of removes increases, our design becomes even more space efficient when compared to the computational CRDT.

6.2 Top Sum

To evaluate our Top Sum design (*NuCRDT*), we compare it against a delta-based CRDT map (*Delta CRDT*) and a state-based computational CRDT implementing the same semantics (*CCRDT*).

The top is configured to display a maximum of 100 entries. In each run, 500000 update operations were generated for 10000 Ids and with challenges awarding scores up to 1000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Figure 3 shows the results of our evaluation. Our design achieves a significantly lower bandwidth cost when compared with the computational CRDT, because in the computational CRDT design, every time the top is modified, the new top is propagated. When compared with the delta-based CRDTs, the bandwidth of NuCRDT is approximately 55% of the bandwidth used by delta-based CRDTs. As delta-based CRDTs also include a mechanism for compacting propagated updates, the improvement comes from the mechanisms for avoiding

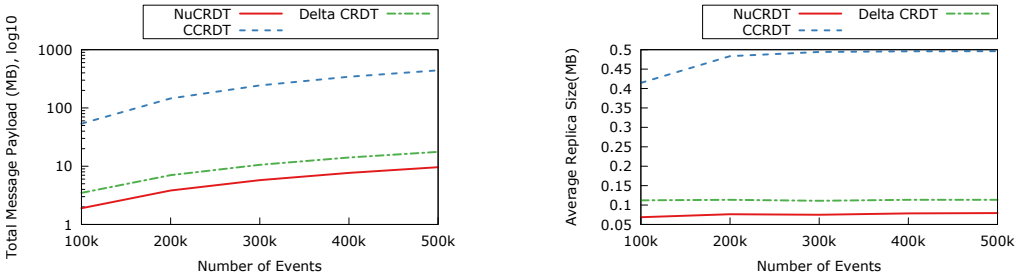


Figure 3 Top Sum: payload size and replica size

propagating operations that will not affect the top elements, resulting in less messages being sent.

The results for the replica size show that our design also manages to be more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information of remote operations received for guaranteeing fault-tolerance and those that have influenced the top elements at some moment in the execution.

7 Conclusions

In this paper we proposed the non-uniform replication model, an alternative model for replication that combines the advantages of both full replication, by allowing any replica to reply to a query, and partial replication, by requiring that each replica keeps only part of the data. We have shown how to apply this model to eventual consistency, and proposed a generic operation-based synchronization protocol for providing non-uniform replication. We further presented the designs of two useful replicated data types, the Top-K and Top Sum, that adopt this model (in appendix, we present two additional designs: Top-K without removals and Histogram). Our evaluation shows that the application of this new replication model helps to reduce the message dissemination costs and the size of replicas.

In the future we plan to study which other data types can be designed that adopt this model and to study how to integrate these data types in cloud-based databases. We also want to study how the model can be applied to strongly consistent systems.

Acknowledgments

This work has been partially funded by CMU-Portugal research project GoLocal Ref. CMUP-ERI/TIC/0046/2014, EU LightKone (grant agreement n.732505) and by FCT/MCT project NOVA-LINCS Ref. UID/CEC/04516/2013. Part of the computing resources used in this research were provided by a Microsoft Azure Research Award.

References

1 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018. doi:10.1016/j.jpdc.2017.08.003.

2 Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proc. 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013. doi:10.1145/2465351.2465361.

3 Gustavo Alonso. Partial database replication and group communication primitives. In *Proc. European Research Seminar on Advances in Distributed Systems*, 1997.

4 Daniel Barbará-Millá and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, July 1994. doi:10.1007/BF01232643.

5 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

6 Tyler Crain and Marc Shapiro. Designing a Causally Consistent Protocol for Geo-distributed Partial Replication. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015. doi:10.1145/2745947.2745953.

7 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 *Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007. doi:10.1145/1294261.1294281.

8 Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, 1987. doi:10.1145/41840.41841.

9 Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulieacute;. Epidemic Information Dissemination in Distributed Systems. *Computer*, 37(5), May 2004. doi:10.1109/MC.2004.1297243.

10 Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics. *Proc. VLDB Endow.*, 9(2):72–83, October 2015. doi:10.14778/2850578.2850582.

11 Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010. doi:10.1145/1773912.1773922.

12 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), July 1978. doi:10.1145/359545.359563.

13 Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2), May 1998. doi:10.1145/279227.279229.

14 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, 2012.

15 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 *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011. doi:10.1145/2043556.2043593.

16 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

17 Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9), July 2013. doi:10.14778/2536360.2536366.

18 Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In

Proceedings of the 26th International Conference on World Wide Web, WWW '17, pages 263–272, 2017. doi:10.1145/3038912.3052603.

19 David Navalho, Sérgio Duarte, and Nuno Preguiça. A Study of CRDTs That Do Computations. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015. doi:10.1145/2745947.2745948.

20 Patrick E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986. URL: http://doi.acm.org/10.1145/7239.7265, doi:10.1145/7239.7265.

21 Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1), March 2005. doi:10.1145/1057977.1057980.

22 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine Partial Replication in Wide Area Networks. In *Proc. 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, 2010. doi:10.1109/SRDS.2010.32.

23 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proc. 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, 2011.

24 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011. doi:10.1145/2043556.2043592.

25 Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proc. 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013. doi:10.1145/2517349.2522731.

26 Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-crdts: Making δ-crdts delta-based. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 12:1–12:4, 2016. doi:10.1145/2911151.2911163.

27 Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1), January 2009. doi:10.1145/1435417.1435432.

A APPENDIX

In this appendix we present two additional NuCRDT designs. These designs exemplify the use of different techniques for the creation of NuCRDTs.

A.1 Top-K without removals

A simpler example of a data type that fits our proposed replication model is a plain top-K, without support for the remove operation. This data type allows access to the top-K elements added to the object and can be used, for example, for maintaining a leaderboard in an online game. The top-K defines only one update operation, *add(id,score)*, which adds element *id* with score *score*. The *get()* operation simply returns the K elements with largest scores. Since the data type does not support removals, and elements added to the top-K which do not fit will simply be discarded this means the only case where operations have an impact in the observable state are if they are core operations – i.e. they are part of the top-K. This greatly simplifies the non-uniform replication model for the data type.

Algorithm 4 Top-K NuCRDT

```
1:  elems : {⟨id, score⟩} : initial {}
2:
3:  GET() : set
4:    return elems
5:
6:  prepare ADD(id, score)
7:    generate add(id, score)
8:
9:  effect ADD(id, score)
10:   elems = topK(elems ∪ {⟨id, score⟩})
11:
12: MASKEDFOREVER(log_local, S, log_recv) : set of operations
13:   adds = {add(id_1, score_1) ∈ log_local : (∃add(id_2, score_2) ∈ log_recv : id_1 = id_2 ∧ score_2 > score_1)
14:   return adds
15:
16: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
17:   return {}                                   ▷ Not required for this data type
18:
19: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
20:   return {add(id, score) ∈ log_local : ⟨id, score⟩ ∈ S.elems}
21:
22: COMPACT(ops): set of operations
23:   return ops                    ▷ This data type does not use compaction
```

Algorithm 4 presents the design of the top-K NuCRDT. The prepare-update *add(id,score)* generates an effect-update *add(id,score)*.

Each object replica maintains only a set of K tuples, *elems*, with each tuple being composed of an *id* and a *score*. The execution of *add(id,score)* inserts the element into the set, *elems*, and computes the top-K of *elems* using the function *topK*. The order used for the *topK* computation is as follows: $\langle id_1, score_1\rangle > \langle id_2, score_2\rangle$ iff $score_1 > score_2 \vee (score_1 = score_2 \wedge id_1 > id_2)$. We note that the *topK* function returns only one tuple for each element *id*.

Function MASKEDFOREVER computes the adds that become masked by other add operations for the same *id* that are larger according to the defined ordering. Due to the way the top is computed, the lower values for some given *id* will never be part of the top. Function MAYHAVEOBSERVABLEIMPACT always returns the empty set since operations in this data type are always core or forever masked. Function HASOBSERVABLEIMPACT returns the set of unpropagated add operations which add elements that are part of the top – essentially, the

add operations that are core at the time of propagation. Function COMPACT simply returns the given *ops* since the design does not require compaction.

A.2 Histogram

We now introduce the Histogram NuCRDT that maintains a histogram of values added to the object. To this end, the data type maintains a mapping of bins to integers and can be used to maintain a voting system on a website. The semantics of the operations defined in the histogram is the following: *add(n)* increments the bin n by 1; *merge(histogram$_{delta}$)* adds the information of a histogram into the local histogram; *get()* returns the current histogram.

Algorithm 5 Histogram NuCRDT

```
 1: histogram : map bin ↦ n : initial []
 2:
 3: GET() : map
 4:   return histogram
 5:
 6: prepare ADD(bin)
 7:   generate merge([bin ↦ 1])
 8:
 9: prepare MERGE(histogram)
10:   generate merge(histogram)
11:
12: effect MERGE(histogram_delta)
13:   histogram = pointwiseSum(histogram, histogram_delta)
14:
15: MASKEDFOREVER(log_local, S, log_recv) : set of operations
16:   return {}                                  ▷ Not required for this data type
17:
18: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
19:   return {}                                  ▷ Not required for this data type
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
22:   return log_local
23:
24: COMPACT(ops): set of operations
25:   deltas = {hist : merge(hist_delta) ∈ ops}
26:   return {merge(pointwiseSum(deltas))}
```

This data type is implemented in the design presented in Algorithm 5. The prepare-update *add(n)* generates an effect-update *merge([$n \mapsto 1$])*. The prepare-update operation *merge(histogram)* generates an effect-update *merge(histogram)*.

Each object replica maintains only a map, *histogram*, which maps *bins* to integers. The execution of a *merge(histogram$_{delta}$)* consists of doing a pointwise sum of the local histogram with *histogram$_{delta}$*.

Functions MASKEDFOREVER and MAYHAVEOBSERVABLEIMPACT always return the empty set since operations in this data type are always core. Function HASOBSERVABLEIMPACT simply returns log_{local}, as all operations are core in this data type. Function COMPACT takes a set of instances of *merge* operations and joins the histograms together returning a set containing only one *merge* operation.

Fine-Grained Consistency Upgrades for Online Services

Filipe Freitas[‡†⋆], João Leitão[†], Nuno Preguiça[†], and Rodrigo Rodrigues[⋆⋆]
[‡]ISEL, Instituto Superior de Engenharia de Lisboa, Portugal; [†]NOVA-LINCS & FCT,
Universidade NOVA de Lisboa, Portugal; [⋆]INESC-ID; [⋆]IST, Universidade de Lisboa, Portugal

***Abstract*—Online services such as Facebook or Twitter have public APIs to enable an easy integration of these services with third party applications. However, the developers who design these applications have no information about the consistency provided by these services, which exacerbates the complexity of reasoning about the semantics of the applications they are developing. In this paper, we show that is possible to deploy a transparent middleware between the application and the service, which enables a fine-grained control over the session guarantees that comprise the consistency semantics provided by these APIs, without having to gain access to the implementation of the underlying services. We evaluated our middleware using the Facebook public API and the Redis datastore, and our results show that we are able to provide fine-grained control of the consistency semantics incurring in a small local storage and modest latency overhead.**

I. Introduction

Many computer systems and applications make use of stateful services that run in the cloud, with various types of interfaces mediating the access to these cloud services. For instance, an application may decide to store its persistent state in a Cassandra cluster running on Azure instances, or directly leverage a cloud storage service such as S3. At a higher level of abstraction, services such as Twitter or Facebook have not only attracted millions of users to their main websites, but have also enabled a myriad of popular applications that are layered on top of those services by leveraging the public APIs they provide.

An important challenge that arises from this layering is that the consistency semantics of these cloud services are almost always not clearly specified, with studies showing that in practice these services expose a number of consistency anomalies to applications [7]. Furthermore, even in the cases where precise specifications exist, it is difficult for programmers to reason about their impact, and this may lead to violations of application invariants that were meant to be preserved [2].

In this paper, we argue that it is possible to build a middleware layer mediating the access to cloud services in order to obtain fine-grained control over the consistency semantics that these services provide. The idea is that we can design a library that intercepts every call to the service or storage system running in the cloud, inserting relevant meta-data, calling the original API, and transforming the results that are obtained in a transparent way for the application. Through a combination of analyzing this meta-data and caching results that have been previously observed, this shim layer can then enforce fine-grained consistency guarantees.

In prior work, Bailis et al. [3] have proposed a similar approach, but with two main limitations compared to this work. First, their shim layer only provides a coarse-grained upgrade from eventual to causal consistency. In contrast, we allow programmers to turn on and off individual session guarantees, where different guarantees have been shown to be useful to different application scenarios [9]. Second, their work assumes the underlying ⟨key,value⟩ store is a NoSQL system with a read/write interface. Such an assumption simplifies the development of the shim layer, since (1) it gives the layer full access to the data stored in the system, and (2) it provides an interface with simple semantics.

In this work, we propose a shim layer that allows for a fine-grained control over the session guarantees that applications should perceive when accessing online services. These services typically enforce rate limits for operations issued by client applications. For guaranteeing that this limit is the same when using our shim layer, a single service operation should be executed for each application operation. Furthermore, our layer is not limited to using online storage services with a read/write interface, since it is designed to operate with services that offer a messaging interface such as online social networks. The combination of these three requirements raises interesting challenges from the perspective of the algorithms that our shim layer implements, e.g., to handle the fact that online social networks only return a subset of recent messages, which raises the question of whether a message does not appear because of a lack of a session guarantee or because of being truncated out of the list of recent messages.

We implemented our shim layer and integrated it with the Facebook API and the Redis storage system. Our evaluation shows that our layer allows for fine-grained consistency upgrades at a modest latency overhead.

The remainder of this paper is organized as follows. Section II discusses our target systems and the assumptions made regarding the centralized system over which third-party applications are developed. Section III discusses the architecture and high level view of our system, while Section IV details the algorithms employed in our solution to enforce each of the session guarantees. Section V discusses our prototype implementation and presents experimental results obtained over two different services. Finally, Section VI discusses relevant related work and Section VII concludes the paper with some final remarks.

978-1-5386-1679-6/17 $31.00 © 2017 IEEE
DOI 10.1109/SRDS.2017.9

IEEE computer society

II. Target systems

Our goal is to provide particular consistency guarantees to third-party applications using popular online web services that expose public APIs. In particular, the application developer may choose to have individual session guarantees (read your write, monotonic reads, monotonic writes, and writes follows reads) as well as combinations of these properties (in particular, all four session guarantees corresponds to causality [5]). To achieve this, we provide a library that can be easily attached to the third-party client application, allowing us to enrich the semantics exposed through the system public API. There are multiple popular systems that provide such public APIs, with various differences in terms of the interface they expose. As such, we needed to focus on a group of APIs with a similar service interface that we can easily adapt to, and we chose to focus on a particular class of services, namely social networks, such as Facebook, Twitter, or Instagram. Our choice is based on the relevance and popularity of these services and also on the large number of third-party applications that are developed for them. In particular, we target services that expose a data model based on key-value stores, where data objects can be accessed through a key, and that associate a list of objects to each key. We observe that this data model is prevalent in online social network services, particularly since they share concepts such as user feeds and comment lists. In particular, we target services where the API provides two fundamental operations to manipulate the list of objects associated with a given key: an insert operation to append a new object to the first position of the list, and a get operation that exposes the first N elements of the list (i.e., the most recent N elements).

Since we access these services through their public APIs, we need to view the service implementation as a black box, meaning that no assumptions are made regarding their internal operation. Furthermore, we design our protocols without making any assumption regarding the consistency guarantees provided through the public service API. The importance of not assuming any guarantees from existing services is justified by our own previous measurement study [7], which showed a high prevalence of violations of multiple session guarantees in public APIs provided by services of this class.

Our algorithms require storing meta-data alongside the data, which can be difficult to do when accessing services as black boxes, namely when the service has no support for including user managed meta-data (this is the case of Facebook, which we explore in the context of our prototype experimental evaluation). In this case, we need to encode this meta-data as part of the data itself. As a consequence, when the service is accessed by native clients (i.e., web applications or third party applications that do not resort to our Middleware) the user might see this meta-data. However, we believe that this is not a crucial issue, since many third party applications only access lists that are used exclusively by that application.

In order to arbitrate an ordering among operations issued by the local client and other remote clients, our Middleware has the need to have an approximate estimate of the current time.

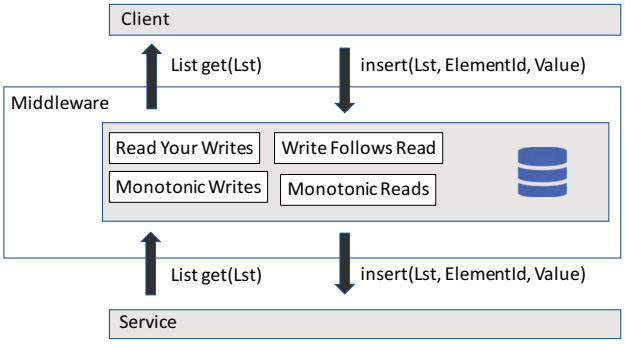


Fig. 1. Middleware

To achieve this, two options are available. If the service has a specific call in its public API that exposes the time in the server, such call can directly be used by our system. Otherwise, if the service exposes a REST API (which is typical in many services) a simple REST call can be performed to the service, and the server time can be extracted from a standard HTTP response header (called *Date*). Note that, even though it is desirable that this estimate is synchronized across clients, we do not require either clock or clock rate synchronization for correctness. In particular, the only negative effect of clocks being out of synch is a reordering of concurrent events from different sessions that is incoherent with their real time occurrence; this can imply, in the case of a service that outputs a sliding window of recent events, that more recent messages may be considered eligible for being truncated (i.e., considered older than the lower end of the window). However, we guarantee that such ordering never violates the correctness conditions we are enforcing.

Finally, we observe that, in practice, the public API exposed by these services often imposes rate limits for operations issued by client applications. These rate limits are exposed under the form of a maximum number of operations that can be executed within a given time window. In particular, we have experimentally observed that violating these rate limits can lead the service to either block further access by the application, or introduce noticeable delays in processing requests issued by the application. The existence of operation rate limits imposes a requirement on our protocols: for each application operation, a single service operation can be issued. This is important to guarantee that an application using our middleware faces the same rate limits as an application using directly the service.

III. System overview

In this section we discuss the general architecture of our solution, which is materialized in a library implementing a middleware layer. We then provide an overview of the operation of our protocols, explaining how they enforce the consistency guarantees of session properties in a transparent way for the client applications.

A. Architecture

Our system consists of a thin layer that runs on the client side and intercepts every call made by the third-party client application on the service, mediating access to the service. In particular, our layer is responsible for contacting the service on behalf of the client application, process the responses returned by the service and generate responses to the client applications according with the session guarantees being enforced. Figure 1 provides a simple representation of this architecture.

Our system can be configured by the third-party application developer to enforce any combination of the individual session guarantees (as defined by Terry et. al [9]), namely: i) read your writes, ii) monotonic reads, iii) monotonic writes, and iv) writes follows reads. In order to enforce these guarantees, our system is required to maintain information regarding previous operations executed by the client application, namely previous writes that were issued or previous values that were observed by the client. In addition, our layer can also insert meta-data that is stored alongside the data in the original system, but stripped by the library before the final response is conveyed to the client.

B. Overview

As mentioned, our system intercepts each request performed by the client application, executes the request in the service, and then processes the answer generated by the service to provide a (potentially different) answer to the client application. This answer is computed based on a combination of the internal state that records the previous operations that were run by that particular client, and the actual response that was returned by the service.

Tracking application activity. In order to keep track of user activity, our system maintains in memory a set of data structures for each part of the service state that is accessed by the application. These data structures are updated according both to the activity of the applications (i.e., the operations that were invoked) and the state that is returned by the service. These data structures are: i) the *insertSet*, which stores the elements inserted by the client and ii) the *localView*, which stores the elements returned to the client.

Enforcing session guarantees. Enforcing session guarantees entails achieving two complementary aspects. First, and depending on the session guarantees being enforced, some additional meta-data must be added when inserting operations. As mentioned, this meta-data can be either added to a specialized meta-data field (if the API exposed by the service allows this) or directly encoded within the body of the element being added to the list. Such meta-data has to be extracted by our library when retrieving the elements of a list, thus ensuring transparency towards client applications. Second, our system might be required to either remove or add elements to the list that is returned by the service when the application issues an operation to obtain the current service state, in order to ensure that the intended session guarantees are not violated.

In the next section, we discuss the concrete algorithms executed by our system upon receiving an insert or get operation for a particular list, in order to ensure that the values observed by the client application adhere to the semantics of each of the session guarantees that are intended to be provided.

Algorithm 1: Initialization of local state

```
1: upon init(Lst) do
2:     lstState ← init()
3:     lstState.insertSet ← {}
4:     lstState.localView ← {}
5:     lstState.lastTimestamp ← 0
6:     lstState.insertCounter ← 0
7:     listStates[Lst] ← lstState
```

IV. Algorithms

We now discuss in more detail the algorithms that are employed by our Middleware layer to enforce session guarantees, and the rationale for their design. To this end, we briefly remind what each of the four session guarantees entails (we extend the definitions previously introduced in [7]), and then explain why our algorithms ensure that the anomalies associated with each of the session guarantees are prevented by it.

We explain our algorithms assuming that the service offers an interface with the following two functions, which are in practice easily mapped to functions that are supported by the various services that we analyzed: the insertion of an element in a given list Lst, denoted by the execution of function `insert`(Lst,$ElementID$,$Value$), where Lst identifies the list being accessed, $ElementID$ denotes the identifier of the element being added (which can be an identifier generated by the centralized service or a unique identifier generated by our Middleware), and $Value$ stands for the value of the element being added to the list; and the access to the contents of a list, denoted by the execution of function `get`(Lst), where Lst identifies the list being read by the client.

When the client accesses a list Lst for the first time, a special initialization procedure is triggered internally by our Middleware (Alg. 1), which initializes the local state regarding the accesses to Lst. The initialization is straightforward: it creates the object *lstState* that maintains all relevant information to manage the accesses to Lst (line 2). This state is composed by the sets **insertSet** and **localView** that were discussed previously, and that are initially empty (lines $3-4$). Furthermore, two other variables are initialized, **lastTimestamp**, which is used to maintain information regarding elements that were removed from the previously discussed sets, and **insertCounter**, which tracks the number of inserts performed by the local client in the context of the current session. Both of these variables have an initial value of zero (lines $5-6$). Finally, the *lstState* variable is stored in a local map, associated to the list Lst (line 7). Next, we explain how this local state is leveraged by our algorithms to enforce the various session guarantees.

A. Read Your Writes

The Read Your Writes (RYW) session guarantee requires that, in a session, any read observes all writes previously

Algorithm 2: Read Your Writes

```
1:  function insert(Lst, ElementId, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.v ← Value
5:      e.id ← ElementId
6:      e.timestamp ← obtainServiceTimeStamp()
7:      SERVICE.insert(Lst, ElementId, e)
8:      lstState.insertSet ← e ∪ lstState.insertSet

9:  function get(Lst) do
10:     lstState ← listStates[Lst]
11:     sl ← SERVICE.get(Lst)
12:     sl ← orderByTimestamp(sl)
13:     sl ← addMissingElementsToSL(sl, lstState.insertSet, lstState.lastTimestamp)
14:     sl ← purgeOldElementFromSL(sl, lstState.insertSet, lstState.lastTimestamp)
15:     lstState.lastTimestamp ← getLastTimestamp(sl)
16:     return removeMetadata(subList(sl, 0, N))
```

Algorithm 3: Monotonic reads

```
1:  function insert(Lst, ElementID, Value) do
2:      Element e ← init()
3:      e.id ← ElementID
4:      e.v ← Value
5:      SERVICE.insert(Lst, ElementID, e)

6:  function get(Lst) do
7:      lstState ← listStates[Lst]
8:      sl ← SERVICE.get(Lst)
9:      lstState.localView ← appendNewElementsToTop(sl, lstState.localView)
10:     return removeMetadata(subList(lstState.localView, 0, N))
```

executed by the same client. More precisely, for every set of insert operations W made by a client c over a list L in a given session, and set S of elements from list L returned by a subsequent get operation of c over L, we say that RYW is violated if and only if $\exists\, x \in W : x \notin S$.

This definition, however, does not consider the case where only the N most recent elements of a list are returned by a get operation. In this case, some writes of a given client may not be present in the result if more than N other insert operations have been performed (by client c or any other client). Considering that the list must hold the most recent writes, a RYW anomaly happens when a get operation returns an older write performed by the client but misses a more recent one. More formally, given two writes x, y over list L executed in the same client session, where x was executed before y, an anomaly of RYW happens in a get that returns S when $\exists\, x, y \in W : x \prec y \wedge y \notin S \wedge x \in S$.

Alg. 2 presents our algorithm for providing RYW. To avoid the anomaly described above, the idea is to store, locally at the client, all elements that are inserted by the local client in the list and add them to the result of get operations. In the insert operation, the inserted element is stored locally by the client (line 8). Additionally, our algorithm stores some meta-data in the object before performing the insert operation over the centralized service (lines $5-6$). This information represents, respectively, the identifier of the element and a timestamp for the insert operation (from the perspective of the service, and retrieved as described in Section II). The element identifier is used to uniquely identify the writes. The timestamp and element identifier allow for totally ordering all entries in the **insertSet**, with the order being approximately that of the real-time order of execution. Note that the operation in line 12 also checks if the timestamps retrieved from the service in the same session are monotonically increasing, and, if not, enforces that property by overwriting the returned timestamp with an increment of the most recent one; this is important to avoid reordering events from the same session in case the timestamp provided by the server does not increase monotonically for some reason.

For executing a get operation (line 9) our algorithm starts by executing the get operation over the service (line 11). Then, the returned list (*sl*) is ordered (line 12) and all elements of the local **insertSet** that are missing in the list are added to the list, keeping it ordered (line 13). Before returning the most recent N elements (with no meta-data) (line 16), our algorithm removes old session elements from the *sl* list and updates the **lastTimestamp** variable with the timestamp of the oldest element of the client session returned to the client (lines $14-16$).

A limitation of this algorithm is that it causes the **insertset** to grow indefinitely. To avoid this, we use the timestamp of each element to remove from the **insertset** any element older than **lastTimestamp**. We also need to include the session id in the metadata of each element to avoid old elements of the session to reaper. We omit this from Alg. 2 for readability.

B. Monotonic Reads

This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. To define this in our scenario where a truncated list of N recent elements is returned, we say that Monotonic Reads (MR) is violated when a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists\, x, y \in S_1 : x \prec y \textit{ in } S_1 \wedge y \notin S_2 \wedge x \in S_2$, where $x \prec y$ means that element x appears in S_1 before y.

To avoid this anomaly, our algorithm (presented in Alg. 3) resorts to the **localView** variable to maintain information regarding the elements (and their respective order) observed by the client in previous get operations. Therefore, when the client issues a get operation, our Middleware issues the get command over the centralized service (line 8) and then updates the contents of its **localView** with any elements that are returned by the service and that were not yet within the **localView** (line 9). These new elements are appended to the start of the list, as they are assumed to be more recent than those of the current **localView**.

The algorithm terminates by returning to the client the N most recent elements in the **localView**. These elements are exposed to the client without any of the meta-data added by our algorithms (line 10). Note that in this case the insert operation only issues the corresponding insert command with additional meta-data on the centralized service (lines $1-5$).

Similar to the previously discussed algorithm, this approach has a limitation regarding the growth of local state, in this

Algorithm 4: Monotonic Writes

```
1:  function insert(Lst, ElementID, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.id ← ElementID
5:      e.v ← Value
6:      e.clientSession ← getClientSessionID()
7:      e.sessionCounter ← lstState.insertCounter++
8:      SERVICE.insert(Lst, ElementID, e)

9:  function get(Lst) do
10:     lstState ← listStates[Lst]
11:     sl ← SERVICE.get(Lst)
12:     sl ← sortElementsBySessionCounters(sl)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     return removeMetadata(sl)
```

Algorithm 5: Write Follows Read

```
1:  function insert(Lst, ElementID, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.id ← ElementID
5:      e.v ← Value
6:      e.cutTimestamp ← obtainCutTimestamp(lstState.localView)
7:      e.dependencies ← projectElementIdentifiers(lstState.localView)
8:      e.timestamp ← obtainIncreasingServiceTimeStamp(lstState.localView)
9:      SERVICE.insert(Lst, ElementID, e)

10: function get(Lst) do
11:     lstState ← listStates[Lst]
12:     sl ← SERVICE.get(Lst)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     cutTimestamp ← highestCutTimestamp(sl)
15:     sl ← removeElementsBelowCutTimestamp(sl, cutTimestamp)
16:     lstState.localView ← appendNewElementsByTimestamp(ls, lstState.localView)
17:     lstState.localView ← purgeOldElements(lstState.localView)
18:     return removeMetadata(sl)
```

case the **localView** can grow indefinitely. To avoid this, we associate with each element inserted in the list a timestamp. This timestamp allows us to remove from the **localView** any element with a timestamp smaller than the timestamp of the oldest element that was in the last return to the client. We omit this from Alg. 3 for readability.

C. Monotonic Writes

This session guarantee requires that writes issued by a given client are observed in the order in which they were issued by all clients. More precisely, if W is a sequence of write operations issued by client c up to a given instant, and S is a sequence of write operations returned in a read operation by any client, a Monotonic Writes (MW) anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W: $\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

However, this definition needs to be adapted for the case where only N elements of a list are returned by a get operation. In this case, some session sequences may be incomplete, because older elements of the sequence may be left out of the truncated list of N returned elements. Thus, we consider that older elements are eligible to be dropped from the output, provided that we ensure that there are no gaps in the session subsequences and that the write order is respected, before returning to the client. Formally, we can redefine MW anomalies as follows, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x, y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

Alg. 4 presents the algorithm employed by our Middleware to enforce the MW session guarantee. We avoid the anomaly described above by adding meta-data to each insert operation (lines $1-8$) in the form of a unique client session id (*clientSession* – line 6) and a counter (local to each client and session) that grows monotonically (*sessionCounter* – line 7). This information allows us to establish a total order of inserts for each client session.

This meta-data is then leveraged during the execution of a get operation (lines $9-14$) in the following way. After reading the current list from the service (line 11), we simply order the elements in the read list (*sl*) to ensure that all elements respect the partial orders for each client session (line 12). Finally, an additional step is required to ensure that no element is missing in any of these partial orders. To ensure this, whenever a gap is found within the elements of a given client session, we remove all elements whose *sessionCounter* is above the one of any of the missing elements.

The get operation returns the contents that are left in the list *sl* without the meta-data added by our algorithms (line 14). Note that in this case we might return to the client a list of elements with a size below N. We could mitigate this behavior by resorting to the contents of the **localView** as we did in the algorithm to enforce MR. However, we decided to provide the minimal behavior to enforce each of the session guarantees in isolation.

D. Write Follows Read

This session guarantee requires that the effects of a write observed in a read by a given client always precede the writes that the same client subsequently performs. (Note that although this anomaly has been used to exemplify causality violations [1], [8], any of the previous anomalies represent a different form of a causality violation [9].) To formalize this definition, and considering that the service only returns at most N elements in a list, if S_1 is a sequence returned by a read invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a violation of the Write Follows Read (WFR) anomaly happens when: $w \in S_2 \wedge \exists x, y \in S_1 : x \prec y$ *in* $S_1 \wedge y \notin S_2 \wedge x \in S_2$.

Our algorithm to enforce this session guarantee is depicted in Alg. 5. The key idea to avoid this anomaly is to associate with each insert the direct list of dependencies of that insert, i.e, all elements previously observed by the client performing the insert (line 7). Evidently, this solution is not practical, since this list could easily grow to include all previous inserts performed during the lifetime of the system. To overcome this limitation, we associate with each insert a timestamp based on the clock of the service, but with the restriction of

being strictly greater than the timestamp of any of its direct dependencies (line 8). Furthermore, we also associate with each insert a cut timestamp, that defines the timestamp of its last explicit dependency, i.e, the dependencies registered in the dependency list (line 6). The cut timestamp implicitly defines every element with a lower timestamp to be a dependency of that insert operation. By combining these different techniques, we ensure that the explicit dependency list associated with an insert has at most a value around N elements (which is the size of the **localView** maintained by our Middleware).

Since only N elements of a list are returned by a get operation, the older dependencies may be left out of the sequence that is returned. When this happens, it is safe to consider that these dependencies were dropped from the window that is returned, provided that we ensure that, for each element that is returned, all dependencies that are more recent than the oldest element are also returned.

In the get operation we leverage this meta-data to do the following: we start by reading the contents of the list from the service (line 12) and then over this list we remove any insert whose dependencies are missing. Thus, we only remove inserts whose missing dependencies have a timestamp above the insert cut timestamp. We then compute a cut timestamp for the obtained list *sl* (line 13) that is the highest cut timestamp among all elements in *sl*. We use this timestamp to remove from *sl* any element whose creation timestamp falls below the computed cut timestamp. Finally, before returning to the client the elements that remain in *sl* without the additional meta-data (line 18) we update and garbage collect old entries from the **localView** (lines $16-17$).

Similarly to the previous algorithm, the service might return a number of elements that is lower than N. In this case, to ensure that we always return N elements, we need to obtain the missing dependencies using a get operation that returns a single element (if supported by the service). In our implementation, we avoided this solution because it is prone to triggering a violation of the API rate limits. Again, an alternative way to address this is by, after reading the list from the service, merging its contents with those in the **localStore** and enforcing an order that is compatible with the timestamp of each element. However, for simplicity in exposition, we omit the details of this alternative.

E. Combining multiple session guarantees

Considering the algorithms to enforce each of the session guarantees discussed above, we can now summarize how to combine them. In a nutshell, it suffices for our Middleware to, on insert operations, add the meta-data used by each of the individual algorithms according to the guarantees configured by the application developer. Correspondingly, upon the execution of a get operation, our Middleware must perform the transformations over the list obtained from the service (*sl*) prescribed by each of the individual algorithms. Furthermore, all meta-data added to each element must also be removed before exposing data to the client application.

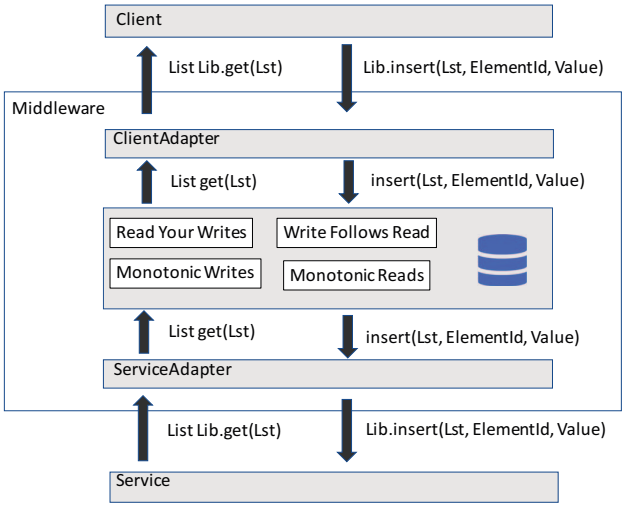


Fig. 2. Middleware with adapters

V. Evaluation

In this section we present the experimental evaluation of our Middleware, which compares the client-perceived performance obtained when using our Middleware to provide each of the session guarantees in isolation and their combination (i.e, enforcing all four session guarantees). In our experiments we used a prototype of our Middleware whose implementation we briefly discuss below. Our evaluation was made using two different geo-replicated online services. First, to illustrate the benefit of our Middleware when designing third-party applications that interact with online social networks we have used Facebook's public API. Then, to illustrate the operation of our Middleware when interacting with a service that imposes fewer restrictions on the number and timing of client operations, we experimented with a geo-replicated deployment of the Redis datastore managed by ourselves.

Our evaluation focuses on asserting the overhead that results from the use of our middleware, in terms of client perceived latency (for insert and get operations), the communication overhead due to the inclusion of additional meta-data, and the storage overhead, namely due to the need for our Middleware to locally maintain some information about previous operations performed by the client.

A. Implementation

Our prototype of the Middleware layer proposed in the paper was implemented in the Java language. To interact with the two services that we explore in this work, we resorted to the *restFB* library for Facebook[1], and the Jedis library for interacting with Redis[2].

Since our prototype was designed to interact with any Internet service with a public API, it requires two adapter layers to be written and provided to its runtime upon execution

[1]http://restfb.com
[2]https://github.com/xetorthio/jedis

(see Figure 2). These layers capture the API calls performed by the client application and translate them to a standard API exposed by our Middleware, and translate the calls to the centralized service performed by our Middleware into API calls to the library used to interact with the service, respectively. We have implemented these adapters for the two case studies employed in this evaluation. The adapters themselves are quite straightforward to write, and we believe most developers will be able to easily write new adapters to use our Middleware in combination with different libraries for accessing other online services.

B. Facebook Results

We have conducted our experiments with Facebook by using YCSB [6] to emulate clients using Facebook to post messages to a group feed and reading the contents of that group feed. To emulate such clients spread across the World, we run three independent YCSB instances in three different locations using Amazon EC2 instances in Oregon, Ireland, and Tokyo. Each YCSB instance uses 10 threads, emulating a total of 30 independent clients, for a total of 90 clients across the World. Each emulated client has an independent instance of our Middleware. To accommodate the rate limits of Facebook's public API, we impose a maximum of 15 requests per second per YCSB instance.

Each experiment reported in this section was executed 7 times, and different consistency guarantees were rotated along experiments, such that each different consistency guarantee had experiments running on different time periods of the day. This was done to remove experimental noise due to contention on the Facebook servers, e.g., due to other user activity. The workload executed by clients was a mix of 50% inserts and 50% gets. The Middleware was configured to have $N = 25$ meaning that each get retrieves at most 25 elements from the feed. Experiments reported in this section report the aggregated observations of $53,119$ insert and get operations.

1) Latency: We start by observing the latency of operations in Facebook when accessing the service directly through the library (labeled in the plots as NONE) and when using our Middleware to enforce each of the session guarantees in isolation and all of the session guarantees (labeled in the plots as ALL).

Figure 3 reports the latency observed for get operations, for all clients and per location of the client. Figure 3(a) shows that our Middleware introduces a small increase in the latency of get operations with a maximum increase of approximately one hundred milliseconds. Not surprisingly the overhead is at its maximum when all session guarantees are being enforced by our Middleware which is explained by a combination of the additional meta-data carried in each element, and the processing cost of the Middleware to perform the enforcement of each individual session guarantee.

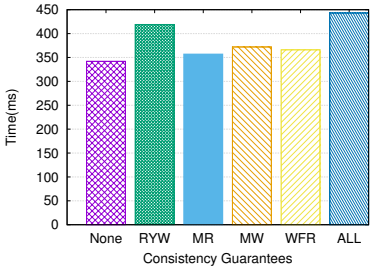
When observing the distribution of latency for requests according to the region where the client is located (Figure 3(b)), we note the same relative distribution in the results, with overall lower latency values for the clients in Oregon. This is explained by the latency of those clients towards the Facebook servers, which is notoriously smaller according to the latency when using the client library directly. Another noteworthy aspect of Figure 3(b) is that the observed latency has a visible variation, both across and even within different client locations. This suggests that the latency overhead in these cases may suffer from a noticeable variability due to external factors which are related with the architecture and deployment of such a large-scale real World application.

Figure 4 reports average latency results for the insert operation for all clients and per client location. The results reported in Figure 4(a) show that globally the latency penalty incurred by the use of our Middleware is again modest, with a maximum increase of at most 50 milliseconds. The individual session guarantee with the largest increase in latency is monotonic reads. Considering the latency values observed in different locations reported in Figure 4(b), we note the same pattern previously observed, where the latency experienced by clients in Oregon is lower compared with the remaining locations. This is expected, since this can be explained by the latency experienced by the client to contact the Facebook service in that concrete location when compared with the remaining locations used in our experimental work.

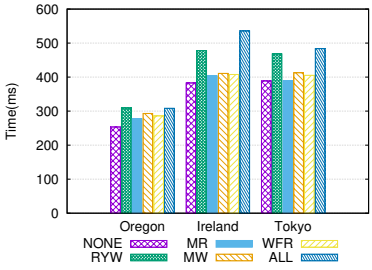
2) Communication Overhead: We now study the communication overhead imposed by our Middleware by observing the average size of messages exchanged between clients and the service. Figure 5 reports these results for each of the session guarantees and for their combination, compared with the use of the library without our Middleware, for both get and insert operations. The results show that the overhead introduced by our Middleware is low for the get operations. This happens because most of the payload in these messages are the multiple elements of the list that are returned. Since our algorithms use small meta-data objects, the communication cost remains dominated by the contents of the elements that are read, as can be observed in Figure 5(a).

The same is not true for insert operations, as reported in Figure 5(b). In this case, since each message contains only a single element to be added, the increase in message size is quite noticeable when the Middleware is enforcing Writes Follows Reads and the combination of all session guarantees. This happens due to the cost of sending the explicit dependencies of each inserted element, which can account to 25 unique element identifiers and their timestamps. The remaining session guarantees, in contrast, have a modest overhead of only a few tens of bytes.

3) Local Storage Size: Finally, Figure 6 reports the storage cost in terms of elements stored locally by our Middleware for enforcing each of the session guarantees and their combination. For completeness, we also provide the results for the NONE configuration, which, as expected, is zero. This is used as a sanity check for our results. Monotonic writes do not require any form of local storage, and therefore have no local storage overhead. In contrast, the remaining session guarantees resort to elements stored in the insertSet and localView data structures. As expected, when providing all of the session

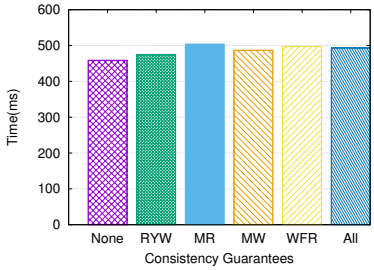


(a) Global

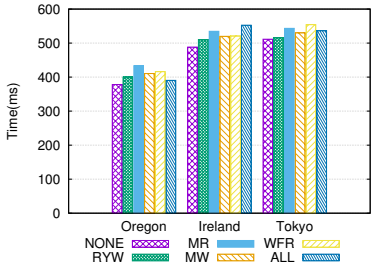


(b) Per location

Fig. 3. Latency of Get Operation in Facebook

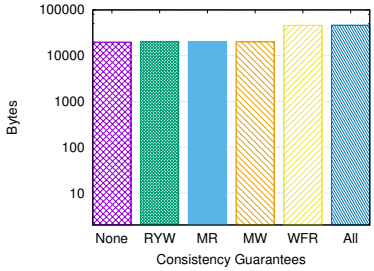


(a) Global

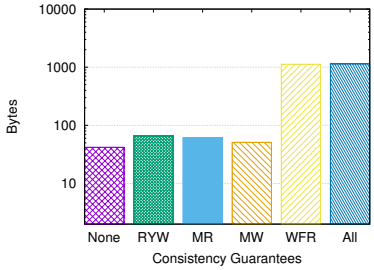


(b) Per location

Fig. 4. Latency of Insert Operation in Facebook



(a) Get Operation



(b) Insert Operation

Fig. 5. Communication overhead in Facebook

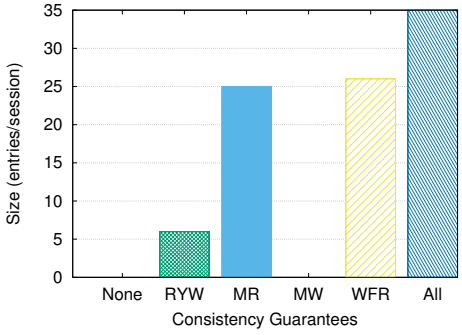


Fig. 6. Local storage overhead for Facebook

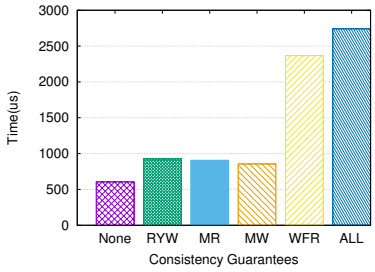
guarantees the local storage has more entries, this happens because the number of entries is the sum of the elements in the insertSet and in the localView.

C. Redis Results

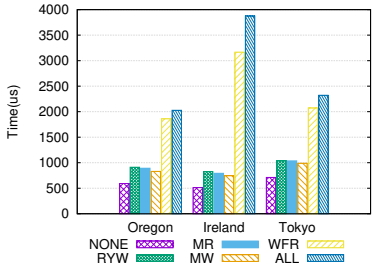
We also conducted experiments using the Redis data storage system. To this end, we have deployed Redis with its replication enabled across machines scattered in three Amazon EC2 regions: Oregon, Tokyo, and Ireland. Redis uses a master-slave replication model, and we have deployed the master in Ireland and two slaves in each region, for a total of 7 replicas. YCSB was executed in the same three regions of Amazon EC2 used in the previously reported experiments, with each YCSB instance running 10 threads that execute operation in a closed loop. Each thread has its own instance of the Middleware. All operations access the same list object stored in Redis, with the read operation being executed in one of the slave replicas, selected randomly. For each algorithm, we run our experiments 6 times for 60 seconds with an interval of four minutes between runs. Similar to the experiments conducted with Facebook, YCSB was configured to execute a workload composed of 50% inserts and 50% of reads. Again, we set N to be equal to 25. The experiments reported in this section aggregate the results from executing a total of $21,285,291$ insert and get operations.

1) Latency: Figure 7(a) presents the average latency of get operations. The results shows that our middleware introduces a very small overhead, on the order of microseconds, for Read Your Writes, Monotonic Reads, and Monotonic Writes. In Write Follows Read and when all session guarantees are enforced, there is an increase of approximately one to two milliseconds because the algorithms have to check the dependencies and process the meta-data. The results of Figure 7(b), which details the values observed in each region, show the same pattern across all regions, namely that the latency for reading data using a client in Ireland is higher than in other locations (due to the proximity to the master replica). This can be explained by the fact that writes in Ireland are much faster that in other locations, which causes the number of read operations that are executed to be higher in Ireland than in other locations, thus leading to a higher load, which results in a higher latency for executing operations.

In contrast to the experiments for the Facebook service, the observed latencies are much more predictable in this deployment. This confirms the expectation that a real-world service leads to qualitatively different results from a controlled experiment.

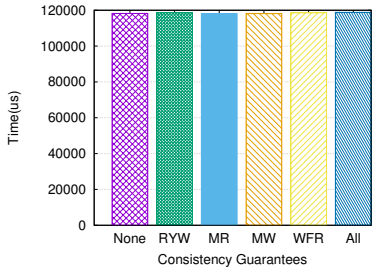


(a) Global

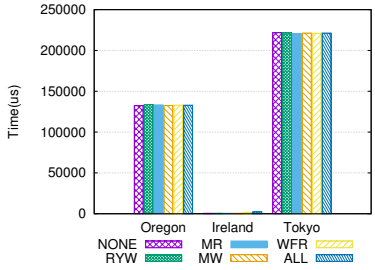


(b) Per location

Fig. 7. Latency of Get Operation in Redis

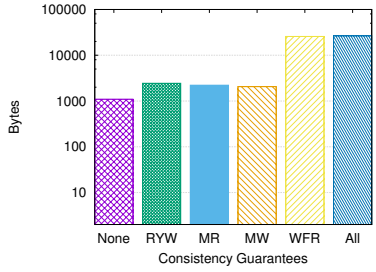


(a) Global

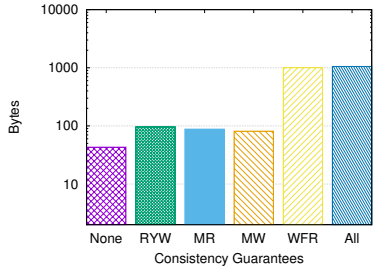


(b) Per Location

Fig. 8. Latency of Insert Operation in Redis



(a) Get Operation



(b) Insert Operation

Fig. 9. Communication overhead in Redis

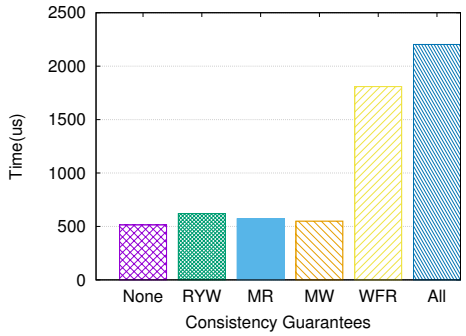


Fig. 10. Latency of Insert Operation in Redis in Ireland

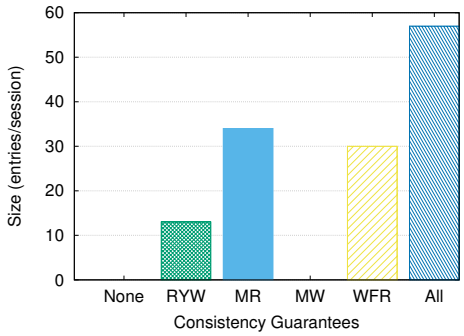


Fig. 11. Local storage overhead for Redis

Figure 8(a) shows the latencies of the insert operation, in this case the latency is almost the same across all cases, but if we look to Figure 8(b) we see that in Ireland latency values are much smaller, this is again justified by the location of the master replica in Ireland and the fact that all clients are issuing their write operations to the (same) master replica. Figure 10 reports the latencies in Ireland, again a similar pattern to the one observed for Get operations.

2) Communication Overhead: In terms of communication overhead imposed by our Middleware, the results in Figure 9(a) and Figure 9(b) show that in the Get and Insert operations the overhead is more noticeable when enforcing Write Follows Read and in when employing the combination of all algorithms. This happens due to the overhead associated with managing and communicating the information stored in dependency lists.

3) Local Storage Size: To conclude, Figure 11 shows that in Monotonic Reads and Write Follows Read the number of elements in the localView is around 30, which is higher than $N = 25$. This happens because of the high write throughput, which causes several elements to be assigned the same timestamp. In this case, our truncation algorithm allows for the limit to be exceeded in the case of ties. The combinations of all algorithms is also affected by this situation, leading to a higher value around 55. Note that we are showing the average of the highest value registered for each independent client session at any time during its execution.

VI. Related Work

The closest related work can be found in the recent proposals that also leverage a middleware layer that can mediate

access to a storage system in order to upgrade the respective consistency guarantees [3], [4].

In particular, Bailis et al. [3] proposed a system called "bolt-on causal consistency" to offer causal consistency on top of eventually consistent data stores. There are two main distinctions between bolt-on causal consistency and our proposal: first, we provide a fine-grained choice of which session guarantees the programmer intends the system to provide, and only pay a performance penalty that is associated with enforcing those guarantees. Second, they assume the underlying system offers a general read/write storage interface, which gives significant more flexibility in terms of the system design than in our proposal, which is restricted to the APIs provided by social networking services.

The other closely related system is the one proposed by Bermbach et al. [4], which is also based on a generic storage interface, namely that provided by S3, DynamoDB, or SimpleDB, in contrast to our focus on high level service APIs. While they also provide fine-grained session guarantees chosen by the programmer, they limit these to Monotonic Reads and Read Your Writes.

Our own prior work provides a measurement study of the violations of session guarantees that are observed when accessing real services [7]. However, the focus of that prior work is on understanding the prevalence of occurrences of lack of session guarantees, whereas this proposal is about fixing those problems through a middleware layer implementing a series of novel algorithms.

VII. Conclusions

In this paper we have shown that it is possible to enforce different consistency properties, in particular session guarantees for third party applications that access online services through their public APIs. We do so without explicit support from the service architecture, and without assuming that the service itself provides any of these guarantees. Our solution relies on a thin Middleware layer that executes on the client side, and intercepts all interactions of the client with the online service. We have presented different algorithms to enforce each of the well known session guarantees. Furthermore, our algorithms follow a simple structure that allows to combine then easily.

We have developed a prototype in Java that we used to evaluate our approach using two centralized services: Facebook, and a geo-replicated deployment of Redis. Our experiments show that we can enforce session guarantees with a modest overhead both in terms of user-perceived latency and communication with the centralized service.

Acknowledgements

This work was partially supported by the EU project LightKone (grant agreement n. 732505) and by FCT (UID/CEC/04516/2013). The research of R. Rodrigues is funded by the European Research Council (ERC-2012-StG-307732) and by FCT (UID/CEC/50021/2013). Part of the computing resources used for this work were supported by an AWS in Education Research Grant.

References

[1] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM.

[3] 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.

[4] David Bermbach, Jorn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13, pages 114–123, Washington, DC, USA, 2013. IEEE Computer Society.

[5] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 152–158, Feb 2004.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[7] F. Freitas, J. Leitao, N. Preguia, and R. Rodrigues. Characterizing the Consistency of Online Services (Practical Experience Report). In *Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 638–645. IEEE, June 2016.

[8] 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.

[9] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

FMKe: a Real-World Benchmark for Key-Value Data Stores

Gonçalo Tomás
NOVA LINCS & DI, FCT,
Universidade NOVA de Lisboa,
Portugal
ga.tomas@campus.fct.unl.pt

Peter Zeller
University of Kaiserslautern,
Germany
p_zeller@cs.uni-kl.de

Valter Balegas
NOVA LINCS & DI, FCT,
Universidade NOVA de Lisboa,
Portugal
v.sousa@campus.fct.unl.pt

Deepthi Akkoorath
University of Kaiserslautern,
Germany
akkoorath@cs.uni-kl.de

Annette Bieniusa
University of Kaiserslautern,
Germany
bieniusa@cs.uni-kl.de

João Leitão
NOVA LINCS & DI, FCT,
Universidade NOVA de Lisboa,
Portugal
jc.leitao@fct.unl.pt

Nuno Preguiça
NOVA LINCS & DI, FCT,
Universidade NOVA de Lisboa,
Portugal
nuno.preguica@fct.unl.pt

ABSTRACT

Standard benchmarks are essential tools to enable developers to validate and evaluate their systems' design in terms of both relevant properties and performance. Benchmarks provide the means to evaluate a system with workloads that mimics real use cases. Although a large number of benchmarks exist for database system, there is a lack of standard benchmarks for an increasingly relevant class of storage systems: geo-replicated key-value stores providing weak consistency guarantees. This has led developers and researchers to rely on ad-hoc tools, whose results are both hard to reproduce and compare.

In this paper, we propose the first standardized benchmark specially tailored for weakly consistent key-value stores. The benchmark, named FMKe, is modeled after a real application: the Danish National Joint Medicine Card. The benchmark is scalable, it can be parameterized to emulate a large number of access patterns, and it is also highly flexible, enabling its application on systems that offer different consistency guarantees and mechanisms.

CCS CONCEPTS

• **General and reference** → *Evaluation*;

KEYWORDS

Benchmark, Key-Value Store

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PaPoC'17, Belgrade, Serbia
© 2017 ACM. 978-1-4503-4933-8/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064889.3064897

ACM Reference format:
Gonçalo Tomás, Peter Zeller, Valter Balegas, Deepthi Akkoorath, Annette Bieniusa, João Leitão, and Nuno Preguiça. 2017. FMKe: a Real-World Benchmark for Key-Value Data Stores. In *Proceedings of PaPoC'17, Belgrade, Serbia, April 23, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3064889.3064897

1 INTRODUCTION

Standard benchmarks provide a uniform way for evaluating and comparing different systems. The most used benchmarks for databases (e.g. TPC-C [1], TPC-W [2], etc.) model realistic applications. As such, this type of benchmarks is expected to provide a more realistic performance evaluation than synthetic benchmarks, where individual operations are generated randomly according to some distribution defined in the workload.

While the TPC-* benchmarks work well for the evaluation of relational, strongly consistent database systems, they are a bad fit for evaluating eventually consistent key-value stores. The main issue is that they do not reflect the way key-value stores are typically used. For example, some aggregation queries in TPC-W are very expensive to implement on top of a key-value data model respecting the specification, which can render the value of experiments useless.

Given the need of evaluating their systems, many system developers opt for implementing their own version of popular applications, like Twitter, FusionTicket [3], or even TPC-C/TPC-W. Yet, there is no standardized way of comparing these ad-hoc implementations due to different codebases and the lack of a common specification. The Yahoo! Cloud System Benchmark (YCSB) [4] addresses this problem by providing a set of standard benchmarks that can be used to evaluate key-value stores. However, the operations of the benchmark consists of simple read/write operations, while real-life applications often use more complex access patterns.

In this paper, we present FMKe, a new application benchmark tailored to the evaluation of key-value stores providing weak consistency. It is based on a subsystem of the Danish National Healthcare

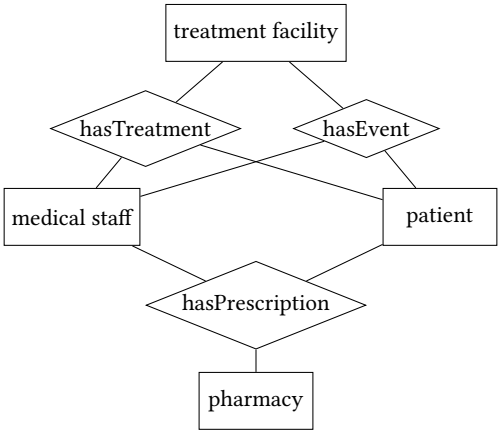


Figure 1: Simplified ER diagram that models FMKe

System (FMK, Fælles Medicinkort), and the workload is defined based on real-life statistics obtained from this production system. FMKe models the handling of prescriptions assigned to patients. This information is accessed concurrently by multiple entities, including medical facilities, such as hospitals, and pharmacies.

The benchmark can be used to evaluate any storage system providing weak consistency guarantees, but also includes variants for evaluating advanced database features, such as highly available transactions.

In the remainder of this paper we present the data model of the FMKe benchmark, describe the operations of the workload, and discuss its implementation and evaluation on top of Antidote [5], a key-value store that supports highly available transactions under geo-replication.

2 FMKE BENCHMARK

In this section we introduce the FMKe benchmark. This benchmark was designed based on a real system that operates at National level in Denmark, which is used to manage medical information, prescriptions, and treatment information for the population of the country. Our benchmark focus on the particular aspect of prescription management and does not include all operations available the the real FMK system.

When designing FMKe, we did not have access to the exact data model used by the original FMK application. Thus, we have designed the data model (presented below) based on the operations provided by FMK to manage prescriptions.

2.1 Data Model

FMKe[6] is a system that manages medical information about patients. In this domain there is a need to keep records for pharmacies, treatment facilities, patients, prescriptions, patient treatments, and medical events (such as taking medicine or medical prognosis updates). The benchmark includes a set of application-level operations, each one of them leading to the execution of a sequence of read and update operations on these entities. The set of hospitals, pharmacies, patients and medical staff act as static entities in the benchmark, so records of these entities can be populated in data stores prior to the benchmark execution (and these can be scaled in number to fit the needs of the system under evaluation). Figure 1 presents a simplified view of the main entities.

2.2 Workload Operations

Table 1 shows the operations performed by the benchmark together with their relative frequency. We have developed two variants of the benchmark with different data layouts. The non-normalized variant follows the strategy of storing data in a denormalized form, which allows to serve most reads without joining data from different records. The other variant stores data in a normalized form, which leads to smaller object sizes, but requires to join data from multiple records when reading. Table 1 includes the respective number of reads and writes for the operations in the two implementations. We now describe the individual operations.

Create prescription registers a new prescription record that is associated with a patient, medical staff and pharmacy. After creation the prescription is considered to be *open* (i.e. it was not yet handled by a pharmacy in order to deliver medicine to the patient).
Process prescription changes the state of a prescription record to signal that it has been handled, so it transitions to the *closed* state.
Get staff prescriptions returns all prescription records that are associated with a specific medical staff member.
Get pharmacy prescriptions returns all prescription records associated with a pharmacy.
Get processed prescriptions returns only prescriptions that have been handled (closed).
Get prescription medication returns the medication for a specific prescription record.
Update prescription medication changes the medication for a prescription that has not been processed.

2.3 Benchmark Characterization

FMKe models as closely as possible the real production system FMK. Benchmark operations and their frequency have default values based in statistics from the real-world system. The benchmark is naturally affected by the number of entities in the data store on which these operations are performed.

Table 2 presents the values for the parameters used in the results presented in the next section – the numbers of hospitals and pharmacies is close to the real numbers, while for patients and doctors the number is between $\frac{1}{3}$ and $\frac{1}{5}$ of the real value. In those experiment, we present results in three different settings where we vary the number of data centers of the deployment.

The benchmarks can be parameterized to use value that match the needs of the system being evaluates, either by changing the number of entities used as well as changing the frequency of each operation.

3 PRELIMINARY EXPERIMENTAL RESULTS

To show the feasibility of the benchmark, we present some preliminary performance results. To this end we have implemented an initial prototype of the benchmark, composed by three components:

Clients The clients issue HTTP/REST requests to the application server, encoding the application operations (section

Operation	Frequency	Non-normalized		Normalized	
		# reads	# writes	# reads	# writes
Get pharmacy prescriptions	27%	1	0	N	0
Get prescription medication	27%	1	0	1	0
Get staff prescriptions	14%	1	0	N	0
Create prescription	8%	5	4	5	4
Get processed pharmacy prescriptions	7%	1	0	N	0
Process prescription	4%	4	4	1	1
Update prescription medication	4%	4	4	1	1

Table 1: Number of read and write operations per FMKe operation. For some operations in the normalized variant the number of reads depends on the current number of prescriptions associated to pharmacy, staff, etc. (denoted by N); this number varies over time.

Entity	Number
Patients	1,000,000
Hospitals	50
Pharmacies	300
Doctors	5,000

Table 2: Number of entities for a workload targeted at performance evaluation

2.2). This module is implemented using Basho Bench [7], an open source benchmarking framework.

FMKe application server The FMKe application server receives client requests, and for each application operation issues a number of operation to modify the state of the database.

Database The data of the benchmark is stored in the database. In our current prototype, we only support Antidote [5].

We ran our experiments in the Amazon Web Service (AWS) infrastructure. Each data center instance consists of four m3.xlarge machines running the Antidote database servers, four m3.xlarge machines running the FMKe application server and four m3.xlarge running the Basho Bench workload generator. A m3.xlarge machine has 4 vCPUs, 15GB of memory and 80 GB of SSD disk. We used the Ireland, Frankfurt and N. Virginia AWS data centers, with the following mean round-trip-time between machines in those data centers: Ireland-Frankfurt: 22.4ms; Ireland-N.Virginia: 84.9ms; Frankfurt-N.Virginia: 89.7ms. The mean round-trip-time between two machines inside a DC was 0.55ms.

Figure 2 shows a throughput-latency plot for the FMKe benchmark on the Antidote system [5]. The plots shows measurements under three different deployments where we vary the number of data centers in each deployment. We based the measurements on a version of FMKe with normalized data layout. The results show that Antidote scales linearly with the number of DCs. The reason for this is that the majority of operations in the workload are read-only. As read-only operations involve only a single DC in Antidote, they do not generate any additional load on the other DCs. Operations that update the database generate additional load when forwarding updates, but the efficient mechanism for update propagation used

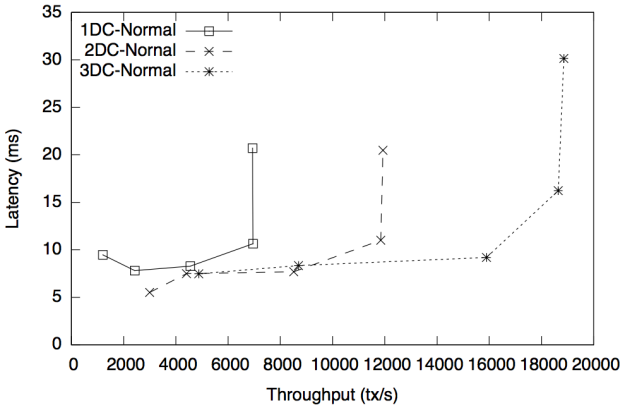


Figure 2: Antidote performance comparison with varying number of data centers.

in Antidote keeps this additional load low, allowing the throughput to almost double when we add the second DC.

Figure 3 shows the detailed results for a single experiment, where it is possible to observe the evolution of throughput and latency during the complete experiment. These graph are generated by Basho Bench, and are very useful to understand the behavior of the system as the database size increases.

4 CONCLUSION AND FUTURE WORK

In this paper we introduced a new benchmark for data stores providing weak consistency, which is modeled after a wide-area healthcare production system for managing medical prescriptions. We briefly presented the data model and operations for this benchmark. We have described our initial prototype and reported preliminary performance results obtained with Antidote database.

As next step we plan to provide a precise specification of the FMKe operations and their functional requirements. From this specification we will derive a reference implementation of the benchmark with bindings for multiple languages. Further, we will define a set of tests that developers can run to assess the consistency and availability properties of their system. For instance, these tests would allow checking whether operations executed atomically, or if the

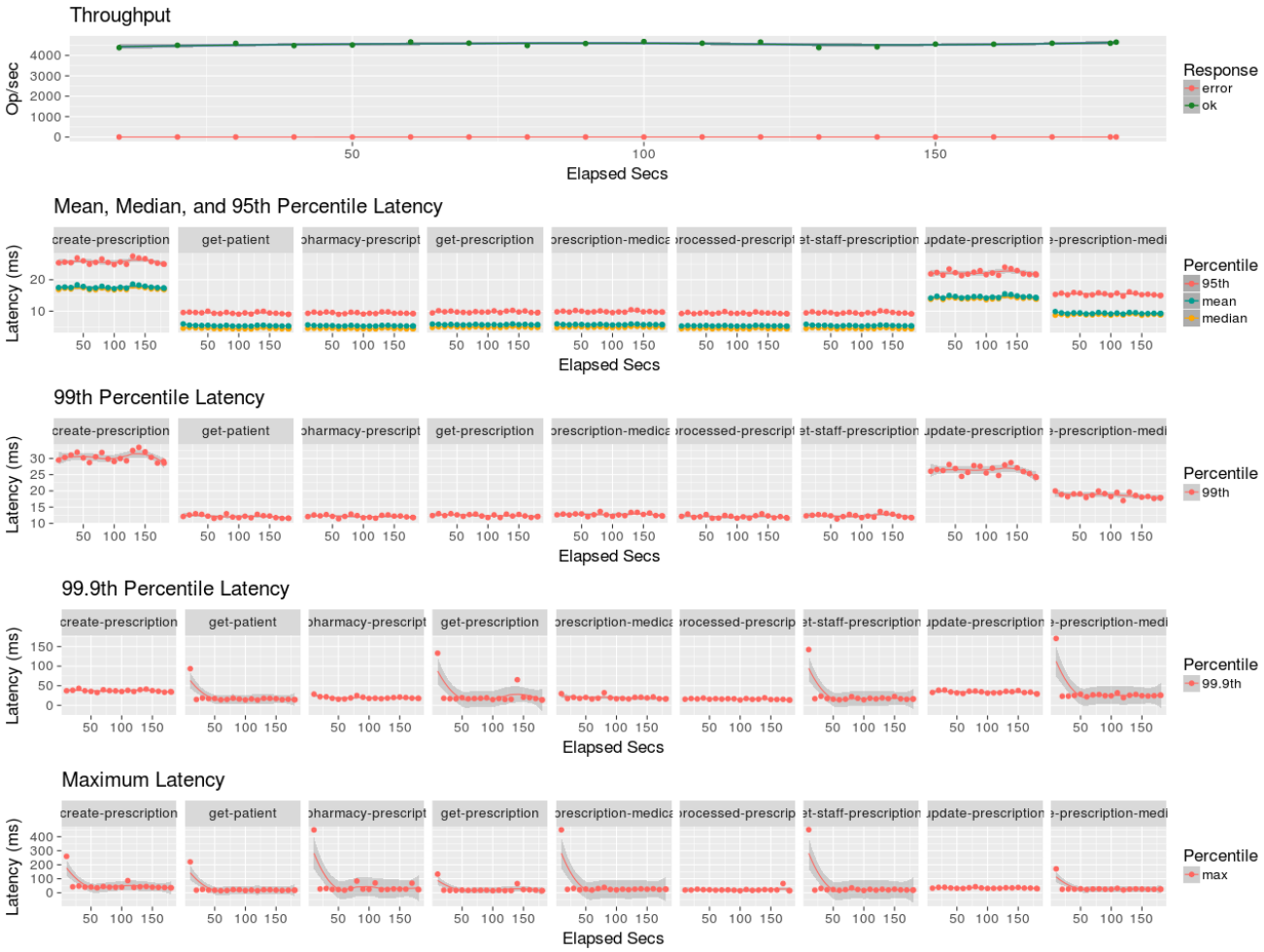


Figure 3: Results for a single experiment (1 DC, 32 clients).

system provides causality. We also aim for mechanisms to measure data staleness, which is a relevant trade-off for storage systems providing weak consistency guarantees and high availability.

Acknowledgements. This work was partially supported by FCT/MCTES: NOVA LINCS project (UID/CEC/04516/2013) and the European Union, through projects SyncFree (grant agreement number 609551) and LightKone (grant agreement number 732505). We would like to thank Kresten Thorup (Trifork) for his help in the design of the benchmark.

REFERENCES

[1] The Transaction Processing Performance Council, Benchmark C. http://www.tpc.org/tpcc/default.asp. Accessed: 2017-02-15.
[2] The Transaction Processing Performance Council, Benchmark C. http://www.tpc.org/tpcw/. Accessed: 2017-02-15.
[3] Fusion Ticket Solutions Limited. https://github.com/fusionticket. Accessed: 2017-02-16.
[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.
[5] AntidoteDB. http://antidotedb.eu. Accessed: 2017-02-15.
[6] FMKe code repository. https://github.com/goncalotomas/fmke. Accessed: 2017-02-15.
[7] Basho Bench. https://docs.basho.com/riak/kv/2.2.0/using/performance/benchmarking/. Accessed: 2017-02-15.

Bringing Hybrid Consistency Closer to Programmers

Gonçalo Marcelino
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa
Portugal

Valter Balegas
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa
Portugal

Carla Ferreira
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa
Portugal

ABSTRACT

Hybrid consistency is a new consistency model that tries to combine the benefits of weak and strong consistency. To implement hybrid consistency, programmers have to identify conflicting operations in applications and instrument them, which is a difficult and error prone task. More recent approaches automatize the process through the use of static analysis over a specification of the application.

In this paper we present a new tool that is under development that tries to make the technology more accessible for programmers. Our tool is based on the same well-founded principles of existing work, but uses an intermediate verification language, Boogie, that improves the tool usability and scope in a number of ways. Using a general language for writing specifications makes specifications easier to write and improves expressiveness. Also, we leverage the language to add a library of CRDTs, which allows the programmer to solve conflicts without coordination. We discuss the features that we have already implemented and how they contribute to improve the technology.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**; **Consistency**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Static verification; replication; integrity invariants

ACM Reference format:
Gonçalo Marcelino, Valter Balegas, and Carla Ferreira. 2017. Bringing Hybrid Consistency Closer to Programmers . In *Proceedings of PaPoC'17, Belgrade, Serbia, April 23, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3064889.3064896

1 INTRODUCTION

Replication is a fundamental technique for achieving better availability, scalability, and fault tolerance in contemporary storage systems. Many of these systems use a combination of *weak* and *strong* consistency models [2, 8, 11], coined as *hybrid consistency* in [6], to coordinate the execution of operations when the correctness of applications is at risk, and leverage the benefits of asynchronous execution when operations are safe.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PaPoC'17, Belgrade, Serbia
© 2017 ACM. 978-1-4503-4933-8/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064889.3064896

To use *hybrid consistency*, programmers need to identify the application invariants that have to be maintained at all times, and instrument the application code to use coordination when necessary. Choosing the "right" coordination is a difficult task. For that, programmers have to reason about the concurrent effects of each operation in the application, in order to determine which operations require coordination, a complex process when dealing with large applications.

More recently a few static analysis tools have been proposed [2, 6, 9] that aid programmers to determine which operations have to be coordinated in order to maintain the application's correctness. These tools receive the specification of an application, and output the pairs of operations that might break the correctness of the application if executed concurrently. With that information, it is possible to derive sets of tokens, that can be associated with operations, to pin-point where in an application the coordination is required. However, existing tools have limitations that constitute a barrier for their adoption by practitioners: they use domain-specific languages not general enough for specifying the behavior of complex applications; they only support basic data types; and, identify conflicts at the grain of operations, leading to overly-conservative executions.

In this paper, we present a new tool that we are developing and discuss how it improves the usability of existing tools in a number of aspects. Our goal is to build a tool that can be used in practice to help programmers build correct applications using a *hybrid consistency* approach. As in [9] our tool automates the proof rule defined and proved sound in [6], ensuring that the coordination generated for a given application is correct.

In our tool the specifications are written in Boogie [3], a versatile intermediate verification language (used by Dafny [7] and VCC [4]). Boogie generates a set of verification conditions, from an input specification, and then uses a STM solver [5] to check those verification conditions. Opting for a verification language means giving the programmer the ability to easily specify more complex behaviors for operations, something lacking in previous works. We provide support for using complex data types, including those provided by the language or specified by the programmer. One particular case, is that we allow specifying conflict-free replicated data types (CRDTs) [10], which can be used to solve conflicting pairs of operations without coordination.

Furthermore we also propose a fine grained approach for operation coordination. Previous tools suggested the coordination of pairs of operations when their execution could potentially invalidate the invariant of the application under scrutiny. Our novel approach takes into account the parameters of these operations, advising operation coordination only for specific combinations of parameters, allowing the pairs of operations to execute concurrently

in all remaining cases. This approach permits more concurrency, while still ensuring the preservation of the application's invariants.

Finally, when the tool detects a conflicting pair, it is capable of providing a counter-example that programmers can use to correct the application.

In the remaining of this paper we present each of these contributions in detail, and discuss the benefits that they provide to the tool.

2 SYSTEM MODEL

We consider a database system composed by a set of objects fully replicated over multiple data centers. An application is defined as a set of high-level operations and a set invariants that express well-formedness rules of the database state. Each operation is defined as a sequence of reads and updates, and has an associated precondition stating the conditions that have to be guaranteed for its safe execution. When an application submits an operation to the local replica, the precondition is checked on the local database state. If the precondition holds, the operation is executed locally, and its effects are propagate asynchronously to remote replicas. Otherwise, the operation has no effect. We assume that the propagation of operation effects respects causality.

We use a token system as the abstract coordination mechanism as defined in [6]. The token system consists of a set of tokens and a symmetric conflict relation over tokens. Each operation may have an associated set of tokens, ensuring that other operations with conflicting tokens cannot be executed concurrently and their execution has to be coordinated.

3 TOOL OVERVIEW

To apply our analysis the programmer has to annotate the application code with a specification. The specification describes the database state, data invariants, preconditions and effects of each operation. To illustrate the static analysis, we use as running example a distributed tournament management application. This application allows the following operations to be executed by its users: $addTournament(t)$ and $remTournament(t)$ register and remove tournament t, respectively, $addPlayer(p)$ registers player p, and $enroll(p, t)$ enrolls player p in tournament t. Additionally, the application is subject to the following integrity invariant: if player p is enrolled in tournament t, both player p and tournament t must be registered.

The input specification is then analysed in three distinct steps.

Safety analysis This first step ensures that none of the specified operations are able to invalidate the application's invariant by executing in standalone manner without any concurrency. This is done to validate the correction of the specification given as input. In our example, if operation $remTournament(t)$ did not have the precondition requiring that no player is enrolled in tournament t, the operation would fail the safety analysis.

Commutativity analysis This second step checks commutativity between all pairs of operations and outputs the subset of these pairs that are not commutative, as well as the sets of tokens needed to address this issue. This is achieved by executing all pairs of operations in both orders and, afterwards, verifying if the state after these executions is the same. In our example, operations $addTournament(t)$ and $remTournament(t)$ do not commute, while $addPlayer(p)$ and $addTournament(t)$ are commutative.

Stability analysis This last step provides the programmer with the set of pairs of operations that cannot execute concurrently, as they can break the application's invariant, as well as the sets of tokens needed to avoid their concurrent execution. This analysis verifies the stability of each operation precondition against all other operations effects. In practice it verifies if the effects of any operation invalidates the precondition of the operation under analysis. As an example, the precondition of $enroll(p, t)$ is not stable under concurrent execution of $remTournament(t)$, while the precondition of $addTournament(t)$ is stable under the effects of $enroll(p, t)$.

4 TOKEN GENERATION

As briefly explained, the tool generates a set of tokens that are used to prevent conflicting pairs of operations from executing concurrently. The tokens that our tool generates are based on the parameters of each operation. This allows more fine-grained concurrency control than previous approaches, which only identify conflicts per operation. More specifically, the tool tests different parameter values for each pair of operations, identifying in which cases different combinations of parameters might invalidate the invariants. We leverage the verification engine to detect efficiently the problematic combination of parameters. The output of the tool is a token system as described before, indicating the relations between conflicting parameters. Taking as an example the pair of operations $enroll(p, t)$ and $remTournament(t)$, the tool outputs that a token must be used if parameter t is the same in the two operations. A complete example is shown in the Appendix A.

Finally, as an alternative to automatically generating the tokens, the programmer can define her own token system. The tool is then able to determine if the provided token system ensures commutativity and stability of the application's operations, while advising the programmer to remove tokens that are not needed to assert these properties, if any. As reducing the number of tokens decreases coordination and leads to a more scalable application

5 CRDT SPECIFICATION

CRDTs are replicated data types that specify well-defined convergence rules that can be used to solve concurrency conflicts. Previous work [2, 6] has demonstrated that these data types can be used to solve some conflicting pairs of operations without using coordination. However, the tools from those works do not provide support for specifying CRDTs.

Our tool allows specifying CRDTs and use them during the analysis process. Moreover, we have defined a library of generic CRDT types that can be used by the programmer. With this library the programmer has the choice between using the tokens or CRDTs, as a way to solve conflicting operations. In the Appendix B we provide a specification of the tournament application that uses a remove-wins CRDT set.

6 CONCLUSION

We have presented a tool to help programmers take full advantage of the hybrid consistency model. The tool is based on intermediate verification language, Boogie, which empowers programmers with

the ability to write general code for specifying applications. We demonstrated the usefulness of the approach by adding support for CRDTs, as an alternative mechanism for solving conflicting pairs. Also, we have extended the existing algorithm for detecting conflicting pairs with support for parameter analysis, which allows more concurrency in applications.

Future work will be focused on giving programmers the ability to certify already existing source code using the specifications given to the tool, as this would allow the programmer to be sure that their specification reflects what is actually implemented. We plan to use available Boogie APIs for general purpose languages, as Java [1] and C [4], to support verification of real code. This is an important aspect for bridging the gap between specification and implementation.

We also plan to explore ways to reduce the annotation effort, since even with a restricted number of case studies some specification patterns emerge. These patterns could be explored to help programmers writing specifications, by automatically generating part of the annotations or at least by proving a set of *best practices* to be followed.

Acknowledgements. This work was partially supported by FCT-MCTES-PT NOVA LINCS project (UID/CEC/04516/2013), FCT/MCT SFRH/BD/87540/2012, EU FP7 SyncFree project (609551), and EU H2020 LightKone project (732505).

REFERENCES

[1] Java parser for the Boogie intermediate verification language. https://github.com/martinschaef/boogieamp. (????). Accessed Feb-2017.

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

[3] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387.

[4] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–42.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[6] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. ACM, New York, NY, USA, 371–384.

[7] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

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

[9] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE Tool: Proving Weakly-consistent Applications Correct. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '16)*. ACM, New York, NY, USA, Article 2, 3 pages.

[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proc. 13th Int. Conf. on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400.

[11] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *Proc. 23d ACM Symp. on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 385–400.

A TOURNAMENT EXAMPLE

This section starts by showing the input Boogie specification[1] for the tournament application. Although the specification code can be verbose a few patterns emerge through the specification code. These patterns could be helpful in reducing the programmer's annotation effort. One such pattern appears in the `ensures` clause that expresses the effects of operations. The update of a state set variable for a given argument is reflected in a `ensures` clause with a `forall` clause stating that the state variable for that argument is updated, while the remaining objects are left unchanged. This part of the clause for the objects not updated could be generated automatically by the tool, reducing the specification effort.

Input specification

```
type Tournament;
type Player;
var enrollment: [Player, Tournament] bool;
var tournaments: [Tournament] bool;
var players: [Player] bool;

function invariant() returns(bool)
{
   forall t: Tournament, p: Player ::
     enrollment[p,t] ==> tournaments[t] && players[p]
}

procedure addTournament(t1: Tournament)
modifies tournaments;
requires true;
ensures forall t: Tournament ::
  t == t1 ==> tournaments[t1] == true
  &&
  t != t1 ==> tournaments[t] == old(tournaments)[t]; { }

procedure remTournament(t1: Tournament)
modifies tournaments;
requires !exists p: Player :: enrollment[p, t1];
ensures forall t: Tournament ::
  t == t1 ==> tournaments[t1] == false
  &&
  t != t1 ==> tournaments[t] == old(tournaments)[t]; { }

// addPlayer and remPlayer can be similarly defined.

procedure enroll(p1: Player, t1: Tournament)
modifies enrollment;
requires players[p1] == true && tournaments[t1] == true;
ensures forall p: Player, t: Tournament ::
  p == p1 && t == t1 ==> enrollment[p1,t1] == true
  &&
  p != p1 || t != t1 ==> enrollment[p,t] == old(enrollment)[p,t]; { }
```

Safety analysis

With the previous specification the safety analysis does not report any errors. However, if we remove the precondition `players[p1] == true` from operation `enroll`, an invariant violation is reported. In this case, Boogie can build a counter-example model to help the programmer in correcting her specification.

[1] For illustrative reasons we removed some parenthesis in clauses `requires` and `ensures`.

```
enrollments -> [Player,Tournament]Bool
players -> [Player]Bool
tournaments -> [Tournament]Bool
p1 -> Player
t1 -> Tournament

Select_[Player,Tournament]Bool -> {
  [Player,Tournament]Bool p1 t1 -> true
  else -> true
}
Select_[Tournament]Bool -> {
  [Tournament]Bool  t1 -> true
  else -> true
}
Select_[Player]Bool -> {
  [Player]Bool p1 -> false
  else -> false
}
```

Stability analysis

```
Conflicting operations:
{ enroll(p, t), remPlayer(p) }
{ enroll(p, t), remTournament(t) }
```

Commutative analysis

Commutativity is only verified for pairs of operations that do not fail the stability analysis.

```
Non-commutative operations:
{ addTournament(t), remTournament(t) }
{ addPlayer(p), remPlayer(p) }
```

Generated tokens

```
enroll(p,t)      : { token_ep(p), token_et(t) }
remTournament(t) : { token_rt(t) }
addTournament(t) : { token_at(t) }
remPlayer(p)     : { token_rp(p) }
addPlayer(p)     : { token_ap(p) }

Conflict relation:
token_ep(p)  : token_rp(p)
token_rt(t)  : token_et(t), token_at(t)
token_at(t)  : token_rt(t)
token_rp(p)  : token_ep(p), token_ap(p)
token_et(t)  : token_rt(t)
token_ap(p)  : token_rp(p)
```

B AVOIDING COORDINATION WITH CRDTS

Generic remove-wins CRDT

```
type Selector;
type CRDTElement = <a>[a]bool;
type CRDT = [Selector]CRDTElement;
const unique add: Selector;
const unique remove: Selector;

axiom( forall s:Selector :: s == add || s == remove );

function CRDTAdd<a>(elem: a, set: CRDT, oldSet: CRDT) returns(bool)
{
  forall e:a :: e == elem ==> set[add][elem] == true
                &&
                e != elem ==> set[add][e] == oldSet[add][e]
  &&
  forall e:a :: set[remove][e] == oldSet[remove][e];
}

function CRDTRemove<a>(elem: a,
                       set: CRDT, oldSet: CRDT) returns(bool)
{
  forall e:a :: e == elem ==> set[remove][elem] == true
                &&
                e != elem ==> set[remove][e] == oldSet[remove][e]
  &&
  forall e:a :: set[add][e] == oldSet[add][e];
}

function CRDTInSet<a>(element: a, set: CRDT) returns(bool)
{
  set[add][element] && !set[remove][element];
}
```

Input specification with CRDT sets

```
type Tournament;
type Player;
var enrollment: [Player, Tournament] bool;
var tournaments: CRDT;
var players: CRDT;

function invariant(enrollment: [Player,Tournament] bool,
                   tournaments: CRDT,
                   players: CRDT) returns(bool)
{
   forall t: Tournament, p: Player ::
     enrollment[p,t] ==> CRDTInSet(p, players)
                         &&
                         CRDTInSet(t, tournaments)
}

procedure addTournament(t1: Tournament)
modifies tournaments;
requires true;
ensures CRDTAdd(t1, tournaments, old(tournaments)); { }

procedure remTournament(t1: Tournament)
modifies tournaments;
requires !exists p: Player :: enrollment[p, t1];
ensures CRDTRemove(t1, tournaments, old(tournaments)); { }

// addPlayer and remPlayer can be similarly defined.

procedure enroll(p1: Player, t1: Tournament)
modifies enrollment;
requires CRDTInSet(p1, players) && CRDTInSet(t1, tournaments);
procedure enroll(p1: Player, t1: Tournament)
ensures forall p: Player, t: Tournament ::
  p == p1 && t == t1 ==> enrollment[p1,t1] == true
  &&
  p != p1 || t != t1 ==> enrollment[p,t] == old(enrollment)[p,t]; { }
```

Stability analysis

```
Conflicting operations:
{ enroll(p, t), remPlayer(p) }
{ enroll(p, t), remTournament(t) }
```

Commutative analysis

All non-conflicting operations are commutative.

Generated tokens

```
enroll(p,t)      : { token_ep(p), token_et(t) }
remTournament(t) : { token_rt(t) }
addTournament(t) : { }
remPlayer(p)     : { token_rp(p) }
addPlayer(p)     : { }

Conflict relation:
token_ep(p)  : token_rp(p)
token_rt(t)  : token_et(t)
token_rp(p)  : token_ep(p)
token_et(t)  : token_rt(t)
```

ACGreGate: A Framework for Practical Access Control for Applications using Weakly Consistent Databases

Mathias Weber[1] and Annette Bieniusa[2]

[1]TU Kaiserslautern, Kaiserslautern, Germany, m_weber@cs.uni-kl.de
[2]TU Kaiserslautern, Kaiserslautern, Germany, bieniusa@cs.uni-kl.de

Abstract

Scalable and highly available systems often require data stores that offer weaker consistency guarantees than traditional relational databases systems. The correctness of these applications highly depends on the resilience of the application model against data inconsistencies. In particular regarding application security, it is difficult to determine which inconsistencies can be tolerated and which might lead to security breaches.

In this paper, we discuss the problem of how to develop an access control layer for applications using weakly consistent data stores without loosing the performance benefits gained by using weaker consistency models. We present ACGreGate, a Java framework for implementing correct access control layers for applications using weakly consistent data stores. Under certain requirements on the data store, ACGreGate ensures that the access control layer operates correctly with respect to dynamically adaptable security policies. We used ACGreGate to implement the access control layer of a student management system. This case study shows that practically useful security policies can be implemented with the framework incurring little overhead. A comparison with a setup using a centralized server shows the benefits of using ACGreGate for scalability of the service to geo-scale.

1 Introduction

The ongoing globalization and digitization of services forces companies to build highly available and scalable applications that operate with low latency anywhere on earth. By their nature, these applications need to be distributed and replicated on a global scale. System designers choose weaker consistency guarantees for applications to gain the required performance and availability. Typically, the data stores underlying these types of systems are

key-value stores where objects are addressed by keys and have a value that can be read and updated. To provide better performance and fail-over for possible outages, the data is replicated to different locations. Because of weaker consistency guarantees offered by the store, operations can be issued on any of the replicas without delay and the modifications of the data state are usually asynchronously sent to the other replicas. Data stores supporting this model are for example Amazon Dynamo[13], Riak KV[6] and Cassandra[1]. But weakening consistency guarantees brings new challenges to the application design and implementation as this model makes it difficult to reason about application correctness.

Access Control One important topic of application design is access control. As definition of access control, we follow the definition provided in prior work[20].

In an application, we distinguish two kinds of operations, data operations and policy operations. Data operations read or modify the application data, and policy operations read or modify permissions of users on objects. Although the need for correct access control is indisputable, implementing it is a daunting task. This is reflected by the OWASP Top 10, an index of the ten security vulnerabilities most often found in web applications. In the OWASP[5] Top 10 from 2013, "Missing Function Level Access Control" was ranked 7th, in the 2017 release candidate, "Broken Access Control" was ranked 4th.

Implementing access control for an application using a weakly consistent datastore is difficult because the requirements of access control seem to be in conflict with the inherent properties of weak consistency. The programmer's intuition is that changes to permissions of a user for an object should be effective immediately in order to provide protection for subsequent data operations which may add sensitive data to the store. Further, changing the access control rules should be possible in order to adapt the application to organizational changes. For example, new employees may enter a company, existing employees may leave. Even the structure of the company might change: departments might get restructured, adding or discharging responsibilities. But despite changes, the decisions on all servers need to be consistent with the current access control rules.

For some applications, access control is deeply embedded in the design of the application. For example, friend lists in social networks provide a mechanism to influence what data becomes visible to other users. A classical example is the social network where Alice added her boss Bob in her friend list, and afterwards she wants to hide her photos of the last party from him. Alice removes Bob from her friend list and afterwards uploads the photos, assuming that Bob does not have access to her photos anymore. But on some server, the updates might get applied in a different order giving Bob

temporary access to the photos. This anomaly leaks data and is the result of the weaker ordering guarantee of the data stores.

As the examples show, the access control system should be dynamic and at the same time consistent to avoid data leakage. These dynamic changes should also be reflected in the decisions to protect the data stored in the system from being leaked. Operations modifying the access control policy can make up a considerable part of the overall operations performed in the system.

Weaker consistency guarantees further result in possible conflicts of concurrent permission assignments. For correct access control, it is often not easy to see how these conflicts can be solved and which conflict-resolution strategies lead to incorrect access control decisions and hence to data leakage.

As an example, we can consider the policy that a consultant may not be responsible for two companies. When instantiating Charly as consultant for company A, someone else might concurrently instantiate Charly as consultant for company B. On the local server, the threat of breaking the security policy may not be detectable since the updates are initially only visible on some other replica. When solving this concurrency conflict, one has to be very careful not to end up in a situation where Charly is consultant for both companies.

Contributions In this paper, we discuss possible architectures to deploy access control in applications using weakly consistent data stores (Section 2). We present requirements for the datastore which make it possible to implement access control correctly, and we introduce ACGreGate, a Java framework for implementing such an access control layer (Section 3). To the best of our knowledge, ACGreGate is the first framework which allows to build access control layers for applications using a weakly consistent datastore. The code of ACGreGate is available at `https://softech-git.informatik.uni-kl.de/mweber/acgregate-java`. Our case study shows that practical applications can be secured using ACGreGate even for complex access control policies with data dependencies (Section 4).

2 Access Control Layer Deployment

In the introduction we have seen that many invariants that are easily implementable under strong consistency are not available in weakly consistent datastores like bounds on values due to the lack of a globally total order on operations. How can we implement access control in environments like that? We can either take a centralized approach where a central server is responsible for keeping the permission data consistent or we can take a distributed approach which is more complex but may yield better performance. In the following, we discuss both approaches in detail.

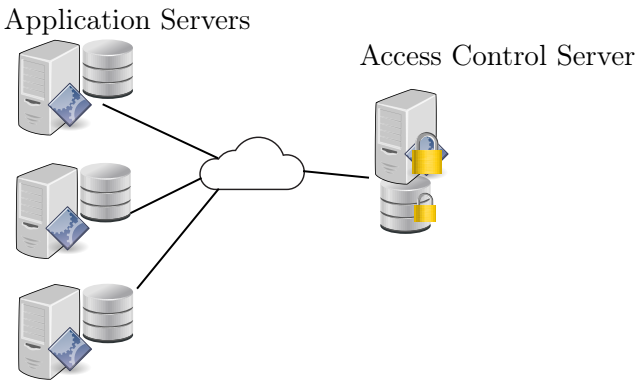


Figure 1: Central access control server architecture.

Central Access Control Server Following the requirement of a consistent access control decision leads to a design as depicted in Figure 1 where a central server makes the access control decision for all distributed copies of the application. This architecture is common in authentication infrastructures. The modifications of the access control rules are handled by the central access control server. All operations performed on remote servers executing the target application are sent to the access control server to make the decision whether the operation should be accepted. This architecture solves the problem of inconsistencies in the access control policy because there is only one copy of the policy on the central server which handles requests in a linearized way. Since all access control decisions are made by this server, we cannot have data leakage because of inconsistent decisions.

The problem with this architecture becomes clear when evaluating the performance (cf. Section 5). Typically, the performance of an application in our scenario is bound by the performance of the storage backend. If the access control server is not running on the same machine as this backend nor is collacated on a fast local network, the performance of the application becomes bound by the network latency. This yields an unacceptable performance overhead compared to a system that does not implement access control. A setup relying on a central access control server is therefore infeasible even for non-large scale systems where delays between nodes are in the order of 10 ms.

Local Consistent Access Control Server What about using a primary backup model for the access control layer (cf. Figure 2)? In this setting, the policy state on all replicas is kept in sync. But the relation between data and policy is not maintained properly. Consider the case that we want to declassify information before publishing it. If we apply the new policy to the old snapshot of the data, we leak classified information. Hence, we need to

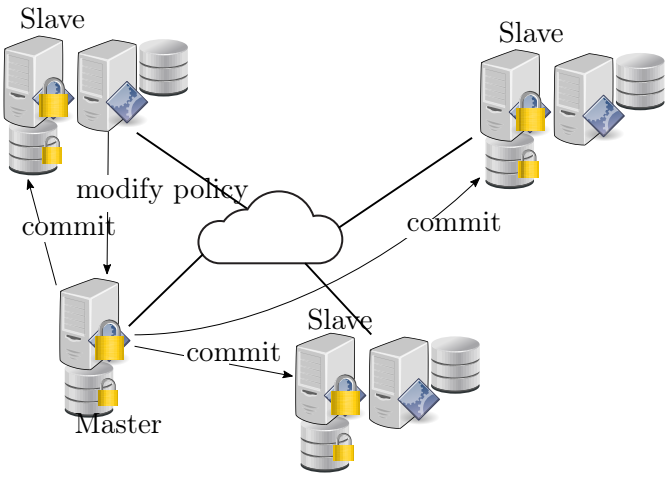


Figure 2: Local consistent access control server architecture.

keep track of all policy modifications and have to apply them consistently to their respective data snapshot.

Another problem is that the strongly consistent policy state reduces the availability. When the application tries to modify the access control policy during a partition of the network, either the master replica or one of the slave replicas is not reachable, thus delaying the execution of the policy modification. Since the application cannot continue without the updated policy, the application is blocked until the network partition is resolved.

Combined Policy and Data State As the discussion shows, there is a strong inherent coupling of permissions and data. We have already shown in prior work[19] that correct access control can be established by putting the policy and data state in the same weakly consistent datastore. Our definition of access control is given in [20] and is based on the fact that policy modifications should become active after they have been issued and until the same policy is modified again. This property can be implemented without reverting to strong consistency or establishing global invariants.

3 Access Control Implementation

The model presented in [19] describes the correctness criteria for access control in weakly consistent information systems according to the above mentioned access control semantics. But there is a considerable gap between the theoretical model and an implementation of this model. In the following, we show how the different correctness criteria can be met by an implementation.

3.1 Requirements derived from the Correctness Condition

The theoretical model considers possible traces of the system. Operations performed by the system need to adhere to the access control policy at the time of the operation execution. Given a concrete trace of a system, this property is straight-forward to verify: We iterate over the operations in the trace and compute the policy based on the policy modification performed and visible at that time and check whether the operation was permitted.

When implementing this model in a real application-level access control system, the task changes. Instead of checking an existing trace, we need to make sure that only operations that are allowed by the current access control policy are actually executed by the system; the operations not permitted should be rejected by the access control layer. This ensures that every resulting trace of the system is valid according to the theoretical model of correct access control.

In order to be able to implement correct access control, we will now derive several prerequisites on the context. The correctness criterium consists of two parts that we will discuss separately: Retaining the protection relation, and correctness of the resolution of conflicts because of concurrent policy modifications.

Retaining the protection relation All policy modifications visible when executing a data modification need to be visible on a replica, before the data operation can be applied. The assumption is that policy modifications can protect subsequent data modifications. By enforcing that the policy modifications is visible before applying that data modification, the model ensures that protected data remains protected and does not leak because of an outdated access control policy. This relation between a policy modification and a subsequent data modification is known as protection relation. As hinted in [19], the protection relation is part of the per-object causality relation. This relation holds between updates u_1 and u_2 on the same object in the data store if u_1 was visible when update u_2 was executed or the other way round. If a data store retains the causality relation such that u_1 is always visible before u_2, we say that the data store is causally consistent. As the example shows, support for causal consistency in the data store is one requirement for being able to implement the access control model correctly.

Causal consistency is known to be the strongest consistency criterion that can still be implemented in a highly-available way. Several data stores implement causal consistency, for example COPS [16], Orbe [14], ChainReaction [8], GentleRain [15], and Antidote [7, 12].

Conflict Resolution The second part of the correctness criterion for access control is concerned with solving the conflicts introduced by concurrent policy modifications in a conservative manner. The policies are modeled in

form of a set of permissions. Each user has for each object in the store a set of permissions on this object. These permission sets may be updated by multiple users concurrently, which can lead to inconsistencies. The required semantics of the data type is already given in [19] in form of the specification of a conflict-free replicated data-type (CRDT). For two sets of permissions s_1 and s_2 concurrently assigned for the same user and the same object, the Policy CRDT takes the intersection of s_1 and s_2. This corresponds to taking only those permissions that both updates agreed on. This semantics can be easily implemented in a data store supporting CRDTs by taking the implementation of a multi-value register and modifying the read operation. A multi-value register retains all concurrently assigned values. Assignments to the register only replace those values visible when the operation is executed on the data store. In case of the Policy CRDT, this yields both sets of permissions that have been assigned concurrently. We can compute the intersection of these sets as the result of the read operation.

3.2 Requirements of an Online Check

In the theoretical model, the decision procedure operates on the same snapshot of data and policies as the operation to be executed. When implementing the model, we have to make sure to preserve this relation between policy and data state. Separating the operations and thereby working on different snapshots of data and policy state can lead to incorrect behavior.

One scenario is that we base our decision on an outdated policy. As already discussed with relation to the theoretical model, policy modifications are usually used to protect data in the store. A policy modification might restrict access to confidential information which was added to the store after the policy modification. If this restriction is not reflected in the *outdated policy*, the access to the confidential information might be granted to an unauthorized user. The other way round, a data modification might have declassified information which is then safe to be read. If we operate on *outdated data*, this data might still reflect the confidential information because the declassifying operations are missing, which again leads to data leakage. Other replicas accept concurrent operations and these operations can in general get visible at the local replica at any time. The conclusion is that there is no safe order in which we can read the policy and execute the operation to be checked. Both operations need to happen atomically.

Atomic operations are supported in some weakly consistent data stores as highly available transactions (HATs) [11, 9, 7]. HATs support atomicity without reducing availability. The main difference to strongly consistent transactions is that HATs do not guarantee serializability of the updates, but the usage of CRDTs solves the problem of conflicting concurrent updates. The Cure protocol[7] for example supports transactional causal+ consistency, an extension of causal consistency with highly available transactions.

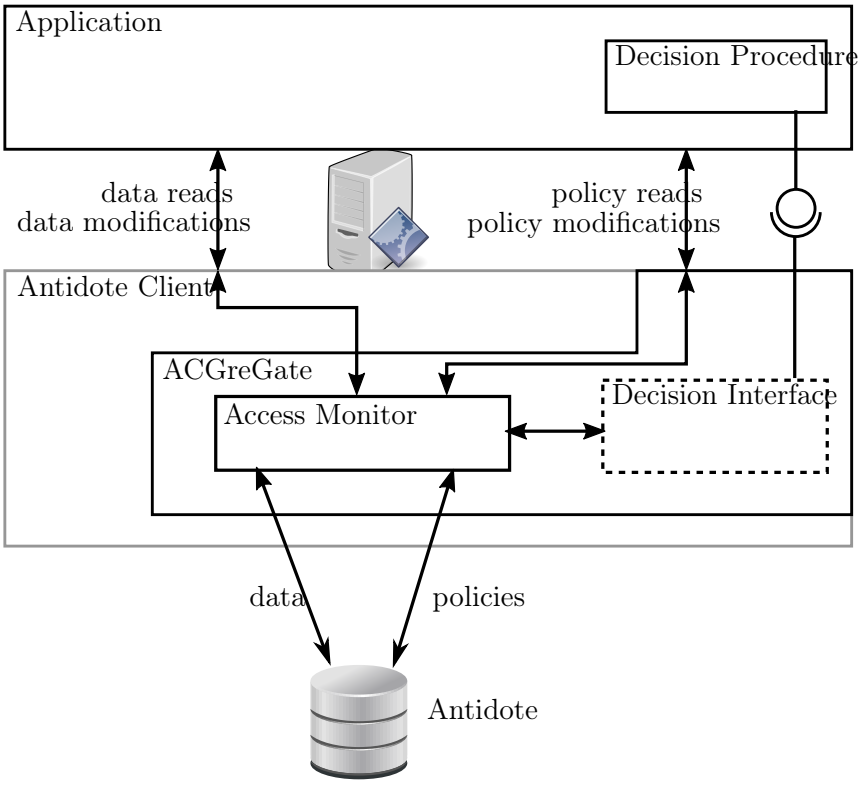


Figure 3: Architecture of ACGreGate.

To summarize, in order to implement our framework for correct access control in applications using weakly consistent data store, we require from the underlying datastore to support causal consistency, CRDTs as data model and transaction support for atomic multi-object updates. All of these properties are supported by the Cure protocol[7].

3.3 Framework Implementation

ACGreGate is a framework for correct access control layers of applications written in Java, that use Antidote as datastore. Antidote[12] is an open-source implementation of the Cure protocol written in Erlang with a variety of client libraries for different programming languages. We built our framework ACGreGate on top of this provided infrastructure for evaluation.

We implemented the Policy CRDT (cf. Section 3.1) and included it in Antidote's CRDT library. Antidote offers a Java client, which allows applications to connect to Antidote and execute update and read operations based on the provided CRDTs. We implemented ACGreGate as an extension to this Antidote Java client library. The architecture of ACGreGate (Figure 3) is described in the following paragraphs in detail.

```
public interface DecisionProcedure {
  boolean decideRead(ByteString currentUser, AntidotePB.ApbBoundObject object,
      Object userData, SecurityLayers layers);

  boolean decideUpdate(ByteString currentUser, AntidotePB.ApbBoundObject
      object, AntidotePB.ApbUpdateOperation op, Object userData, SecurityLayers
      layers);

  boolean decidePolicyRead(ByteString currentUser, AntidotePB.ApbBoundObject
      key, ByteString user, Object userData, SecurityLayers layers);

  boolean decidePolicyAssign(ByteString currentUser, AntidotePB.ApbBoundObject
      key, ByteString user, Collection<ByteString> newPolicy,
      Collection<ByteString> oldPolicy, Object userData, SecurityLayers layers);

  LayerDefinition requestedPolicies(ByteString currentUser,
      AntidotePB.ApbBoundObject object);
}
```

Listing 1: The DecisionProcedure interface.

Decision Procedure To implement the online check, ACGreGate intercepts all operations sent by the client library to the Antidote database and processes them by an access control monitor. This monitor takes an implementation of a decision procedure to make the access control decisions. The decision procedure is application dependent and corresponds to the security policy of the application. In principle, the decision procedure takes the operation, the currently acting user, and the permissions of this user and decides, whether the operation is allowed to be executed or not. In the framework implementation, the decision procedure is implemented as a class implementing the DecisionProcedure interface which can be seen in Listing 1.

The decision procedure is split up for convenience into methods to decide about read and update operations as well as policy read and policy assign operations. Depending on the type of operation, different parameters are provided to the decision procedure, most notably the user performing the operation and information about the operation itself. Access to the current permissions of the acting user is provided using the concept of *security layers*. The concept behind security layers is best described based on an example.

In our model, each user has a single set of permissions on an object. In practice, many applications have hierarchical data structures and users are given permissions on different layers of this hierarchy. A simple example is a university with lecturers. A lecturer gives a lecture and has full access to all data regarding this lecture. A teaching assistant supervises the exercise of the lecture. Tutors may be employed to mentor a group of students in a particular exercise session. Regarding the data of a student participant,

users may have permissions from different levels. A lecturer has the permission to access the data of any participant of the exercise if she reads the corresponding lecture. An assistant of the exercise has permission to access the participant's data if he is the assistant of the participant's exercise. Finally, a tutor may access the participant's data only if she is the participant's tutor. In the datastore, this hierarchy can be split into objects with key references. If we would only allow a single permission set per user and object, we would have to copy all permissions on the lecture to the exercise of this lecture and copy all permissions of the exercise to all participants of this exercise. For registering new participants, we would have to set the permissions on the lecture and exercise, as well. This approach is error prone and complex. Instead, we allow to define the hierarchy for an object in form of a LayerDefinition.

The definition of a layer consists of a name for the layer and the key of the object the layer refers to. A LayerDefinition can contain multiple layers. The definition of the layers is transformed into a SecurityLayers object which allows access to the permissions of the currently acting user on all objects defined in the the LayerDefinition. The complete set of permissions for a user on an object can be computed by taking the union of the permissions on all layers. This approach allows to avoid redundancies in the permission assignments, but still gives access to the complete set of permissions in a well-defined way.

Operation Constraints The key-value data stores we consider support CRDTs which are in essence data types that can safely be used in an unsynchronized distributed manner. The operations on a CRDT object are typically on a higher level than in strongly consistent datastores. For example, Counter CRDTs allow to increment or decrement the counter and to read the current value, Set CRDTs allow to add and remove elements and Map CRDTs allow to update the value of a specific key in the map. Access control policies on the data level of such a datastore usually restrict these operations and describe which properties the operations should have.

For maps, we might want to restrict which keys should be updatable and which key bindings should be removable. For sets, we might want to restrict which elements should be addable or removable. ACGreGate supports implementing these constraints in the implementation of a decision procedure by providing a domain specific language to describe these restrictions. An example of such constraints can be seen in Listing 2.

The constraint specifies that the operation needs to be a map update (isMapUpdate(...)). The additional constraints are that only the key "participants" may be updated and no key-value mappings may be removed (noMapRemoves). The update of the key "participants" is again restricted to be a set update (isSetUpdate(...)). Additional constraints regarding the update of the partic-

```
AntidotePB.ApbUpdateOperation op = ...;
String studentid = ...;

boolean mayParticipate = isMapUpdate(and(
  assignsOnly("participants"),
  constrainAssigns(keyConstrain("participants",
    isSetUpdate(
      or(
        and(
          setAddsOnly(studentid),
          noSetRemoves
        ),
        and(
          setRemovesOnly(studentid),
          noSetAdds
        )
      )))),
    noMapRemoves)).appliesTo(op);
```

Listing 2: Constraint language examples.

ipant set are that either the student identification number may be added to the set (setAddsOnly(studentid)) and no elements may be removed from the set (noSetRemoves), or the other way round. This constraint can be checked against the operation provided as parameter to the decision procedure (.appliesTo(op)). In the context of the case study presented in Section 4, the semantics of the constraint is that students can add themselves as participants to an exercise group.

Datastore Layout To guarantee atomicity and causal consistency of data and policy updates, we persist the permission sets together with the data in the datastore. Because the permissions should only be modifiable using special access control operations, these sets need to be isolated from the other data in the data store. Antidote supports buckets as name spaces for keys in the key value store. These buckets are used in ACGreGate to achieve the isolation between policy and data state. The bucket of the intercepted operations are prefixed with distinct prefixes depending on wether the operation is a data or a security operation. The permission sets are saved per object and user by generating a key based on the object key and the user identifier. This approach guarantees complete isolation between policy and data state and avoids accidental or malicious modification of the permission sets.

Interface Modifications To allow easy replacement of the standard client library with ACGreGate, we kept the interface as close as possible to the original client interface. Only minor modifications with respect to the trans-

action interface are needed in order to specify the currently acting user. When starting a new transaction, the identifier representing the current user has to be passed as a parameter. All operations executed in this transaction are executed under the name of this user. All other modifications are done by returning subtypes of the original client library types such that ACGreGate can act as a substitute to the original client library. The type AntidoteClient needs to be substituted by SecureAntidoteClient, the type Bucket needs to replaced with SecuredBucket.

Data Access In practice, access control policies may depend on the data state of the application. A concrete example of an access control policy that needs this feature is self-registration for an exercise in a lecture management system. Students are allowed to register themselves for an exercise while the exercise registration is open. The flag that signals that the registration is currently open is usually part of the data state of the application, yet it has influence on the access control decision. In this case, the decision procedure implementation will access the object value of the exercise layer and check the status flag for the exercise registration.

This feature enables additional and powerful policies which enable to provide also ownership and attribute-based access control. To implement an ownership model of objects in the database, one could associate a register-type attribute with each object that holds the current owner of the object. The decision procedure can access this attribute and decide on the concrete permissions.

ACGreGate supports policies involving data state by giving the decision procedure access to the current value of a layer object. However, policies involving data state are not covered by our theoretical model in [19] and as such, there is no guarantee that enforcing these policies avoids data leakage in all cases. Instead, the safety of the policy with respect to data leakage has to be reasoned about on a per-application and policy level.

4 Case Study

We show the applicability of ACGreGate to practical access control policies by implementing the access control layer of a student achievement tracking system (STATS). In the following, we describe the data structure and the access control policy of STATS.

4.1 Data Model

The purpose of STATS is to manage the grading of exercises associated to lectures together with the corresponding exams. A student can register an account which includes personal data such as the name, the student identification number, and an email address. At the begin of each semester,

students can register for participation in an exercise. The status of the exercise registration is indicated by a flag in the exercise object. The students registered for an exercise are organized into groups for which a time slot on a specific day is allocated on which a class is given in a specified location. One or more assistants are responsible for the exercise organization. The exercise classes are given by tutors which are responsible for several exercise groups. Students can obtain points for homework submissions which are persisted in form of a map from sheet id to the achieved points.

Similarly, students can register and participate in exams. An exam can be an open exam, for example a trial exam, which is indicated by a flag, or it is only open for students who participated successfully in the exercises. In the end, the exam results together with the grade assignment are published which is indicated again by a flag in the corresponding exam object.

4.2 Access Control Policy

The STAT System was developed in the context of maintaining the integrity of structured data [17]. The access control policy of the STAT System was formulated in terms of integrity constraints. Therefore, only updates were originally restricted.

The access control policies for the update operations are defined as follows. Admins are allowed to perform all updates. Assistants of an exercise are allowed to modify the attributes of the exercise as well as open and close the exercise registration. They are also allowed to create and update groups and sheets of the exercise as well as delete them. Additionally, assistants are allowed to assign students to groups and take them out again as well as to assign and remove tutors. Tutors are allowed to assign students of their group to teams and to modify the results of students in their group. Examiners can modify the attributes of the exam and can add and remove participants. They can create, modify and delete tasks and grades and assign results to participants. Examiners can also open the exam for registration for students and publish the results of the exam for the participants. Students can always register for an exercise, but they can only sign-up for or sign-out of a group if the exercise registration is open. Students can also register for and unregister from exams that are open to self-registration. We now added further the access control rule that assistants and tutors can see the results of their groups and that examiners can only see the results of their own exams. Students have only access to their personal results of the exercises and exams.

These policy rules are based on the application functions. To be implemented as a decision procedure in ACGreGate, they have to be translated to rules on the data update level. For example, an update of an exercise object is allowed if the user is an admin, an assistant of this exercise or a student such that the exercise adds the student id to the set of registered students

for the exercise. We did this for all rules stated above and implemented a decision procedure according to these rules. The result is a Java method with less then 250 lines of Java including error handling and logging. Even though the rules include dependencies to data values, the system is secure in the sense that it avoids data leakage:

- The exercise and exam registration flags control, whether students can register themselves; these flags do not expose additional data.
- Setting the publish flag in the exam exposes the exam results to the students. Assigning this flag to different values concurrently might expose the exam results without consensus. However, we argue that this situation does not arise in practice. Typically, a single examiner sets the flag on behalf of all examiners once they decided to publish the results.

5 Evaluation and Results

The STAT System has been in active use in the CS department at the University of Kaiserslautern in Germany since 2011, though the current implementation is based on a different data persistence layer. We captured the data state of roughly two years with four exercise iterations and six exams. A workload generator takes this data and recreates the same data state in the Antidote backend of STATS. Since the application performs consistency checks, this workload triggers 102 861 datastore read operations and 32 991 datastore update operations. The execution of the decision procedure triggers additional read operations for retrieving the current policy and additional data for some rules. This yields additional 216 923 read operations, which is an overhead of about 135% over the actual operation. The tests we report on here were performed on a server with two Intel Xeon E5-2620 CPUs and 64 GB of RAM running Ubuntu 16.04.1 LTS. We used Docker to build images of the Antidote database and the STAT System. The Docker version used was 1.12.6.

We performed two types of tests. In the first setup, we started one container with the STAT System with the access control layer implemented using ACGreGate. This container was connected to a container running Antidote using a Docker virtual network. The second setup used a modified version of the STAT System that forwarded the access control decisions to an access control server implementing the STATS policy running as a separate container. The STATS container was connected to the Antidote container using one virtual network and connected to the access control server using another separate virtual network. On this network, we used the netem network emulator to simulate different latencies between the application and the access control server.

Table 1: Performance results.

	net delay	**request delay**	**throughput**	**duration**
ACGreGate	-	-	1 079.2 ops/s	2:06
nodelay	0 ms	0.3 ms	1 257.9 ops/s	1:48
small	10 ms	13.9 ms	75.9 ops/s	29:51
medium	50 ms	69.0 ms	16.3 ops/s	2:19:11
large	100 ms	136.8 ms	8.3 ops/s	4:34:21

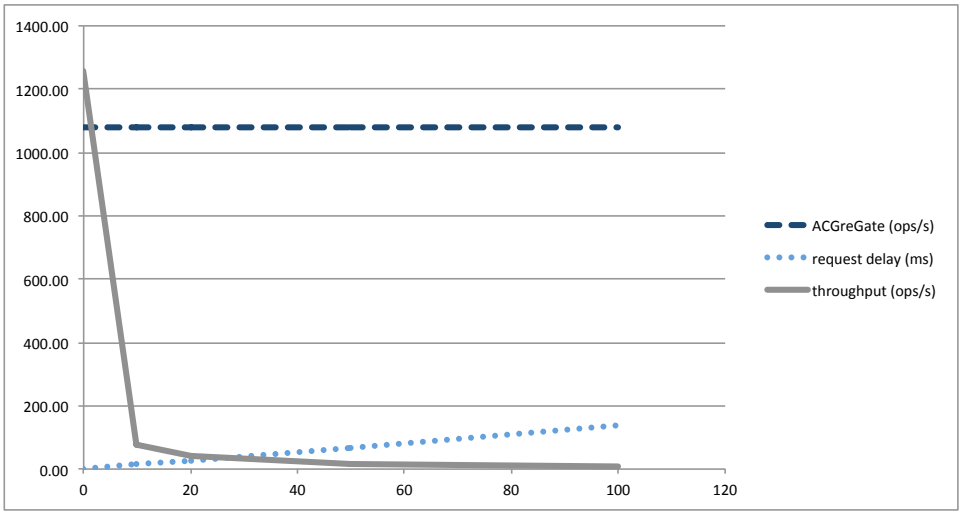


Figure 4: Performance comparison for different network delays.

The results of these experiments are given in Table 1. The net delay refers to the emulated delay of the network between the application and the access control server. The throughput is calculated for the operations issued by the application excluding the operations produced by the decision procedure. The throughput in the ACGreGate and nodelay tests are bounded only by the performance of a single Antidote node. When adding network delay, this delay dominates the performance. Even with only 10 ms roundtrip delay, the performance drops from almost 1 300 operations per second to about 76 operations per second. This delay is roughly observed for servers in different nearby cities in the same country. For more than 50 ms of roundtrip delay on the network, the performance drops by two orders of magnitude to about 16 operations per second, which makes using a central access control server infeasible for cross-country or even geo-scale replication.

The graph in Figure 4 shows a comparison between the performance on ACGreGate and the performance of using a central access control server. ACGreGate has direct access to the policy and gets an immediate response for reading the permissions from the local state in the Antidote datastore. This makes the performance independent of the network delay. In contrast,

the centralized architecture is very sensitive to network delay. For a network delay of 0 ms, the centralized architecture performs slightly better then ACGreGate in our test. The reason for this is that we used a very fast in-memory data store to implement the access control server whereas ACGreGate reads the policies from Antidote. Using a persistent database, for example Antidote, to implement the access control server would yield a result equivalent to the performance of ACGreGate for the 0 ms delay case.

When running the workload generator without access control checks enabled, the throughput is at roughly 1750 ops/s and the program runs in 1:17 minutes, which shows a slowdown for ACGreGate of about 40 %. This slowdown is not unexpected and depends on the number of requests for additional information required by the decision procedure. The performance could further be improved by co-locating data and security attributes or caching of access control decisions.

There are access control architectures that influence the performance far less but also offer less security guarantees. A role-based access control system can be setup with a central authentication server that also manages the roles of users. When authenticating to the server, the user opens a session for which the roles of the user are active that were valid when performing the authentication. This setup can make access control decisions locally without contacting the server again and without reading additional data from the store. But modifications of user permissions do not become effective together with the data modifications, but only after the next authentication request of the user. In addition, role-based access control is far more restrictive with respect to the security policies than the model used by ACGreGate.

6 Related Work

Access Control for Applications using Weakly Consistent Data Stores The topic of access control for applications using weakly consistent data stores has received surprisingly little attention. The original version of Amazon Dynamo [13] did not offer authentication and authorization capabilities. Several other related eventually consistent data stores offer meanwhile techniques to implement access control, but the granularity is not fine enough to provide access control on the application level. Riak KV [6], MongoDB [3], Couchbase [2] and Cassandra [1] all support the management of users, roles and permissions. But the smallest granularity is on the level of buckets or collections, comparable with tables in relational data bases. Typical permissions on this level allow to read, write, modify, or delete any value of the bucket or collection. A more fine-grained permission level relating to the operations on the application level is not supported. MongoDB Stich [4] is a framework for applications built on MongoDB that provides support for access control. The policies supported are mainly based on the current

data state and correctness of the access control system is not clear for any definition of access control.

Samarati et. al[18] describe a high-level approach to authorization in eventually consistent systems. The general idea is to optimistically accept all operations and compensate the operations which were executed despite the security policy by performing rollbacks. While this approach guarantees convergence of the security policy, it is not clear for each operation how to undo the effect of this operation after it has been executed. One of the problems is the potential binding between operations and effects in the data store and changes of the real world. For example, a banking system allows to withdraw money from an account and the ATM outputs the money. In this case, it is hard to undo the withdrawal because the person with the money has already walked away. In addition, the guarantees given by such an optimistic system remain unclear. Effects of operations can be perceived by a user of the data store before the rollback, thereby possibly leaking sensible information.

Wobber et. al[21] present an access control model for weakly consistent, mutually distrustful replicated systems. Their focus of work is on partial replication with different access policies per replica. While we consider a different setting of fully replicated systems, similar problems can be identified. The causality between a policy and the subsequent operation that are permitted by the policy is captured in their model by waiting for the required policy change to arrive. However, the causality between a policy change that restricts the visibility of the effect of an operation and the subsequent execution of this operation is not captured. As such, the model still allows leaking sensitive information because of the possible violation of the protection relation between a policy change and a subsequent data operation.

Access Control for Distributed Applications In the area of access control for distributed applications, most of the work applies to strongly consistent systems. Bui et. al[10] developed an algorithm named FACADE for fast evaluation of stateful attribute-based access control policies. The policies supported by the algorithm are more expressive than the policies we consider. In their work, access control decisions can be influenced by prior access control decisions which requires a linear order of operations and additional state to coordinate the distributed access control decisions. The performance impact of the evaluation is too large for the high-performance applications we consider, the author give an average latency of about 37 ms for their algorithm. In addition to that, the coordination needed to support stateful access control policies reduces the availability of the application as described in Section 2.

6.1 Conclusion

Providing correct, low-latency access control is a challenge for developers of highly available and scalable systems. A careful analysis of the intricate interplay of data and policies shows that such an access control system can be based on causally consistent data stores with support for highly available transactions and conflict resolution of concurrent updates. To show the feasibility of such a system, we implemented ACGreGate, a Java framework for applications based on the Antidote data store. Our case study shows that ACGreGate allows to implement non-trivial policies encountered in systems in active use today. As the use case shows, using ACGreGate to implement access control in an application limits the scalability and performance only to the scalability and performance of the datastore.

References

[1] Apache Cassandra, August 2017. URL: `http://cassandra.apache.org/`.

[2] Couchbase, August 2017. URL: `http://www.couchbase.com/`.

[3] MongoDB for GIANT Ideas – MongoDB, August 2017. URL: `https://www.mongodb.org/`.

[4] MongoDB Stitch – Backend as a Service, August 2017. URL: `https://www.mongodb.com/cloud/stitch`.

[5] Owasp - the free and open software security community, August 2017. URL: `https://www.owasp.org/`.

[6] Riak KV, August 2017. URL: `http://basho.com/products/riak-kv/`.

[7] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 405–414, June 2016.

[8] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[9] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. 7(3):181–192.

[10] Thang Bui, Scott D. Stoller, and Shikhar Sharma. Fast Distributed Evaluation of Stateful Attribute-Based Access Control Policies. In *Data and Applications Security and Privacy XXXI*, Lecture Notes in Computer Science, pages 101–119. Springer, Cham.

[11] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually Consistent Transactions. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 67–86.

[12] SyncFree Consortium. Antidote reference platform, August 2017. URL: `https://github.com/SyncFree/antidote`.

[13] 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.

[14] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[15] 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.

[16] 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.

[17] Patrick Michel. *A formal framework for maintaining the integrity of structured data.* Dr. Hut, 2014.

[18] Pierangela Samarati, Paul Ammann, and Sushil Jajodia. Maintaining replicated authorizations in distributed database systems. *Data & Knowledge Engineering*, 18(1):55 – 84, 1996.

[19] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. *Access Control for Weakly Consistent Replicated Information Systems*, pages 82–97. Springer International Publishing, Cham, 2016.

[20] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. *EPTL - A Temporal Logic for Weakly Consistent Systems (Short Paper)*, pages 236–242. Springer International Publishing, Cham, 2017.

[21] Ted Wobber, Thomas L. Rodeheffer, and Douglas B. Terry. Policy-based access control for weakly consistent replication. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 293–306, New York, NY, USA, 2010. ACM.

Non-uniform Replication

Gonçalo Cabrita[1] and Nuno Preguiça[2]

1 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
g.cabrita@campus.fct.unl.pt
2 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
nuno.preguica@fct.unl.pt

Abstract

Replication is a key technique in the design of efficient and reliable distributed systems. As information grows, it becomes difficult or even impossible to store all information at every replica. A common approach to deal with this problem is to rely on partial replication, where each replica maintains only a part of the total system information. As a consequence, a remote replica might need to be contacted for computing the reply to some given query, which leads to high latency costs particularly in geo-replicated settings. In this work, we introduce the concept of non-uniform replication, where each replica stores only part of the information, but where all replicas store enough information to answer every query. We apply this concept to eventual consistency and conflict-free replicated data types. We show that this model can address useful problems and present two data types that solve such problems. Our evaluation shows that non-uniform replication is more efficient than traditional replication, using less storage space and network bandwidth.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Non-uniform Replication; Partial Replication; Replicated Data Types; Eventual Consistency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.0

1 Introduction

Many applications run on cloud infrastructures composed by multiple data centers, geographically distributed across the world. These applications usually store their data on geo-replicated data stores, with replicas of data being maintained in multiple data centers. Data management in geo-replicated settings is challenging, requiring designers to make a number of choices to better address the requirements of applications.

One well-known trade-off is between availability and data consistency. Some data stores provide strong consistency [5, 17], where the system gives the illusion that a single replica exists. This requires replicas to coordinate for executing operations, with impact on the latency and availability of these systems. Other data stores [7, 11] provide high-availability and low latency by allowing operations to execute locally in a single data center eschewing a linearizable consistency model. These systems receive and execute updates in a single replica before asynchronously propagating the updates to other replicas, thus providing very low latency.

With the increase of the number of data centers available to applications and the amount of information maintained by applications, another trade-off is between the simplicity of maintaining all data in all data centers and the cost of doing so. Besides sharding data among multiple machines in each data center, it is often interesting to keep only part of the data in each data center to reduce the costs associated with data storage and running


© Gonçalo Cabrita and Nuno Preguiça;
licensed under Creative Commons License CC-BY

protocols that involve a large number of replicas. In systems that adopt a partial replication model [22, 25, 6], as each replica only maintains part of the data, it can only locally process a subset of the database queries. Thus, when executing a query in a data center, it might be necessary to contact one or more remote data centers for computing the result of the query.

In this paper we explore an alternative partial replication model, the non-uniform replication model, where each replica maintains only part of the data but can process all queries. The key insight is that for some data objects, not all data is necessary for providing the result of read operations. For example, an object that keeps the top-K elements only needs to maintain those top-K elements in every replica. However, the remaining elements are necessary if a remove operation is available, as one of the elements not in the top needs to be promoted when a top element is removed.

A top-K object could be used for maintaining the leaderboard in an online game. In such system, while the information for each user only needs to be kept in the data center closest to the user (and in one or two more for fault tolerance), it is important to keep a replica of the leaderboard in every data center for low latency and availability. Currently, for supporting such a feature, several designs could be adopted. First, the system could maintain an object with the results of all players in all replicas. While simple, this approach turns out to be needlessly expensive in both storage space and network bandwidth when compared to our proposed model. Second, the system could move all data to a single data center and execute the computation in that data center or use a data processing system that can execute computations over geo-partitioned data [10]. The result would then have to be sent to all data centers. This approach is much more complex than our proposal, and while it might be interesting when complex machine learning computations are executed, it seems to be an overkill in a number of situations.

We apply the non-uniform replication model to eventual consistency and Conflict-free Replicated Data Types [23], formalizing the model for an operation-based replication approach. We present two useful data type designs that implement such model. Our evaluation shows that the non-uniform replication model leads to high gains in both storage space and network bandwidth used for synchronization when compared with state-of-the-art replication based alternatives.

In summary, this paper makes the following contributions:

- The proposal of the non-uniform replication model, where each replica only keeps part of the data but enough data to reply to every query;
- The definition of non-uniform eventual consistency (NuEC), the identification of sufficient conditions for providing NuEC and a protocol that enforces such conditions relying on operation-based synchronization;
- Two useful replicated data type designs that adopt the non-uniform replication model (and can be generalized to use different filter functions);
- An evaluation of the proposed model, showing its gains in term of storage space and network bandwidth.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 describes the non-uniform replication model. Section 4 applies the model to an eventual consistent system. Section 5 introduces two useful data type designs that follow the model. Section 6 compares our proposed data types against state-of-the-art CRDTs.

2 Related Work

Replication: A large number of replication protocols have been proposed in the last decades [8, 27, 15, 16, 2, 21, 17]. Regarding the contents of the replicas, these protocols can be divided in those providing full replication, where each replica maintains the full database state, and partial replication, where each replica maintains only a subset of the database state.

Full replication strategies allow operations to concurrently modify all replicas of a system and, assuming that replicas are mutually consistent, improves availability since clients may query any replica in the system and obtain an immediate response. While this improves the performance of read operations, update operations now negatively affect the performance of the system since they must modify every replica which severely affects middle-scale to large-scale systems in geo-distributed settings. This model also has the disadvantage of limiting the system's total capacity to the capacity of the node with fewest resources.

Partial replication [3, 22, 25, 6] addresses the shortcomings of full replication by having each replica store only part of the data (which continues being replicated in more than one node). This improves the scalability of the system but since each replica maintains only a part of the data, it can only locally process a subset of queries. This adds complexity to the query processing, with some queries requiring contacting multiple replicas to compute their result. In our work we address these limitations by proposing a model where each replica maintains only part of the data but can reply to any query.

Despite of adopting full or partial replication, replication protocols enforce strong consistency [17, 5, 18], weak consistency [27, 7, 15, 16, 2] or a mix of these consistency models [24, 14]. In this paper we show how to combine non-uniform replication with eventual consistency. An important aspect in systems that adopt eventual consistency is how the system handles concurrent operations. CRDTs have been proposed as a technique for addressing such challenge.

CRDTs: Conflict-free Replicated Data Types [23] are data types designed to be replicated at multiple replicas without requiring coordination for executing operations. CRDTs encode merge policies used to guarantee that all replicas converge to the same value after all updates are propagated to every replica. This allows an operation to execute immediately on any replica, with replicas synchronizing asynchronously. Thus, a system that uses CRDTs can provide low latency and high availability, despite faults and network latency. With these guarantees, CRDTs are a key building block for providing eventual consistency with well defined semantics, making it easier for programmers to reason about the system evolution.

When considering the synchronization process, two main types of CRDTs have been proposed: state-based CRDT, where replicas synchronize pairwise, by periodically exchanging the state of the replicas; and operation-based CRDTs, where all operations need to be propagated to all replicas.

Delta-based CRDTs [1] improve upon state-based CRDTs by reducing the dissemination cost of updates, sending only a delta of the modified state. This is achieved by using *delta-mutators*, which are functions that encode a delta of the state. Linde et. al [26] propose an improvement to delta-based CRDTs that further reduce the data that need to be propagated when a replica first synchronizes with some other replica. This is particularly interesting in peer-to-peer settings, where the synchronization partners of each replica change frequently. Although delta-based CRDTs reduce the network bandwidth used for synchronization, they continue to maintain a full replication strategy where the state of quiescent replicas is equivalent.

Computational CRDTs [19] are an extension of state-based CRDTs where the state of the object is the result of a computation (e.g. the average, the top-K elements) over the executed updates. As with the model we propose in this paper, replicas do not need to have equivalent states. The work we present in this paper extends the initial ideas proposed in computational CRDTs in several aspects, including the definition of the non-uniform replication model, its application to operation-based eventual consistency and the new data type designs.

3 Non-uniform replication

We consider an asynchronous distributed system composed by n nodes. Without loss of generality, we assume that the system replicates a single object. The object has an interface composed by a set of read-only operations, $\mathcal{Q}$, and a set of update operations, $\mathcal{U}$. Let $\mathcal{S}$ be the set of all possible object states, the state that results from executing operation o in state $s \in \mathcal{S}$ is denoted as $s \bullet o$. For a read-only operation, $q \in \mathcal{Q}$, $s \bullet q = s$. The result of operation $o \in \mathcal{Q} \cup \mathcal{U}$ in state $s \in \mathcal{S}$ is denoted as $o(s)$ (we assume that an update operation, besides modifying the state, can also return some result).

We denote the state of the replicated system as a tuple $(s_1, s_2, \ldots, s_n)$, with s_i the state of the replica i. The state of the replicas is synchronized by a replication protocol that exchanges messages among the nodes of the system and updates the state of the replicas. For now, we do not consider any specific replication protocol or strategy, as our proposal can be applied to different replication strategies.

We say a system is in a quiescent state for a given set of executed operations if the replication protocol has propagated all messages necessary to synchronize all replicas, i.e., additional messages sent by the replication protocol will not modify the state of the replicas. In general, replication protocols try to achieve a convergence property, in which the state of any two replicas is equivalent in a quiescent state.

▶ **Definition 1** (Equivalent state)**.** Two states, s_i and s_j, are *equivalent*, $s_i \equiv s_j$, iff the results of the execution of any sequence of operations in both states are equal, i.e., $\forall o_1, \ldots, o_n \in \mathcal{Q} \cup \mathcal{U}, o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$.

This property is enforced by most replication protocols, independently of whether they provide strong or weak consistency [13, 15, 27]. We note that this property does not require that the internal state of the replicas is the same, but only that the replicas always return the same results for any executed sequence of operations.

In this work, we propose to relax this property by requiring only that the execution of read-only operations return the same value. We name this property as *observable equivalence* and define it formally as follows.

▶ **Definition 2** (Observable equivalent state)**.** Two states, s_i and s_j, are *observable equivalent*, $s_i \overset{\circ}{\equiv} s_j$, iff the result of executing every read-only operation in both states is equal, i.e., $\forall o \in \mathcal{Q}, o(s_i) = o(s_j)$.

As read-only operations do not affect the state of a replica, the results of the execution of any sequence of read-only operations in two observable equivalent states will also be the same. We now define a non-uniform replication system as one that guarantees only that replicas converge to an observable equivalent state.

▶ **Definition 3** (Non-uniform replicated system)**.** We say that a replicated system is non-uniform if the replication protocol guarantees that in a quiescent state, the state of any two replicas is observable equivalent, i.e., in the quiescent state $(s_1, \ldots, s_n)$, we have $s_i \overset{\circ}{\equiv} s_j, \forall s_i, s_j \in \{s_1, \ldots, s_n\}$.

3.1 Example

We now give an example that shows the benefit of non-uniform replication. Consider an object *top-1* with three operations: (i) $add(name, value)$, an update operation that adds the pair to the top; (ii) $rmv(name)$, an update operation that removes all previously added pairs for *name*; (iii) $get()$, a query that returns the pair with the largest value (when more than one pair has the same largest value, the one with the smallest lexicographic name is returned).

Consider that $add(a, 100)$ is executed in a replica and replicated to all replicas. Later $add(b, 110)$ is executed and replicated. At this moment, all replicas know both pairs.

If later $add(c, 105)$ executes in some replica, the replication protocol does not need to propagate the update to the other replicas in a non-uniform replicated system. In this case, all replicas are observable equivalent, as a query executed at any replica returns the same correct value. This can have an important impact not only in the size of object replicas, as each replica will store only part of the data, but also in the bandwidth used by the replication protocol, as not all updates need to be propagated to all replicas.

We note that the states that result from the previous execution are not equivalent because after executing $rmv(b)$, the get operation will return $(c, 105)$ in the replica that has received the $add(c, 105)$ we operation and $(b, 100)$ in the other replicas.

Our definition only forces the states to be observable equivalent after the replication protocol becomes quiescent. Different protocols can be devised giving different guarantees. For example, for providing linearizability, the protocol should guarantee that all replicas return $(c, 105)$ after the remove. This can be achieved, for example, by replicating the now relevant $(c, 105)$ update in the process of executing the remove.

In the remainder of this paper, we study how to apply the concept of non-uniform replication in the context of eventually consistent systems. The study of its application to systems that provide strong consistency is left for future work.

4 Non-uniform eventual consistency

We now apply the concept of non-uniform replication to replicated systems providing eventual consistency.

4.1 System model

We consider an asynchronous distributed system composed by n nodes, where nodes may exhibit fail-stop faults but not byzantine faults. We assume a communication system with a single communication primitive, $mcast(m)$, that can be used by a process to send a message to every other process in the system with reliable broadcast semantics. A message sent by a correct process is eventually received by all correct processes. A message sent by a faulty process is either received by all correct processes or none. Several communication systems provide such properties – e.g. systems that propagate messages reliably using anti-entropy protocols [8, 9].

An object is defined as a tuple $(\mathcal{S}, s^0, \mathcal{Q}, \mathcal{U}_p, \mathcal{U}_e)$, where $\mathcal{S}$ is the set of valid states of the object, $s^0 \in \mathcal{S}$ is the initial state of the object, $\mathcal{Q}$ is the set of read-only operations (or *queries*), $\mathcal{U}_p$ is the set of prepare-update operations and $\mathcal{U}_e$ is the set of effect-update operations.

A query executes only at the replica where the operation is invoked, its source, and it has no side-effects, i.e., the state of an object remains unchanged after executing the operation.

When an application wants to update the state of the object, it issues a prepare-update operation, $u_p \in \mathcal{U}_p$. A u_p operation executes only at the source, has no side-effects and generates an effect-update operation, $u_e \in \mathcal{U}_e$. At source, u_e executes immediately after u_p.

As only effect-update operations may change the state of the object, for reasoning about the evolution of replicas we can restrict our analysis to these operations. To be precise, the execution of a prepare-update operation generates an instance of an effect-update operation. For simplicity, we refer the instances of operations simply as operations. With O_i the set of operations generated at node i, the set of operations generated in an execution, or simply the set of operations in an execution, is $O = O_1 \cup \ldots \cup O_n$.

4.2 Non-uniform eventual consistency

For any given execution, with O the operations of the execution, we say a replicated system provides *eventual consistency* iff in a quiescent state: (i) every replica executed all operations of O; and (ii) the state of any pair of replicas is equivalent.

A sufficient condition for achieving the first property is to propagate all generated operations using reliable broadcast (and execute any received operation). A sufficient condition for achieving the second property is to have only commutative operations. Thus, if all operations commute with each other, the execution of any serialization of O in the initial state of the object leads to an equivalent state.

From now on, unless stated otherwise, we assume that all operations commute. In this case, as all serializations of O are equivalent, we denote the execution of a serialization of O in state s simply as $s \bullet O$.

For any given execution, with O the operations of the execution, we say a replicated system provides *non-uniform eventual consistency* iff in a quiescent state the state of any replica is observable equivalent to the state obtained by executing some serialization of O. As a consequence, the state of any pair of replicas is also observable equivalent.

For a given set of operations in an execution O, we say that $O_{core} \subseteq O$ is a set of core operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet O_{core}$. We define the set of operations that are irrelevant to the final state of the replicas as follows: $O_{masked} \subseteq O$ is a set of masked operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

▶ **Theorem 4** (Sufficient conditions for NuEC). *A replication system provides* non-uniform eventual consistency (NuEC) *if, for a given set of operations O, the following conditions hold: (i) every replica executes a set of core operations of O; and (ii) all operations commute.*

Proof. From the definition of core operations of O, and by the fact that all operations commute, it follows immediately that if a replica executes a set of core operations, then the final state of the replica is observable equivalent to the state obtained by executing a serialization of O. Additionally, any replica reaches an observable equivalent state. ◀

4.3 Protocol for non-uniform eventual consistency

We now build on the sufficient conditions for providing *non-uniform eventual consistency* to devise a correct replication protocol that tries to minimize the operations propagated to other replicas. The key idea is to avoid propagating operations that are part of a masked set. The challenge is to achieve this by using only local information, which includes only a subset of the executed operations.

Algorithm 1 presents the pseudo-code of an algorithm for achieving *non-uniform eventual consistency* – the algorithm does not address the durability of operations, which will be discussed later.

Algorithm 1 Replication algorithm for non-uniform eventual consistency

```
1:  S : state: initial s^0                                          ▷ Object state
2:  log_recv : set of operations: initial {}
3:  log_local : set of operations: initial {}                       ▷ Local operations not propagated
4:
5:  execOp(op): void                                                ▷ New operation generated locally
6:    log_local = log_local ∪ {op}
7:    S = S • op
8:
9:  opsToPropagate(): set of operations ▷ Computes the local operations that need to be propagated
10:   ops = maskedForever(log_local, S, log_recv)
11:   log_local = log_local \ ops
12:   opsImpact = hasObservableImpact(log_local, S, log_recv)
13:   opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
14:   return opsImpact ∪ opsPotImpact
15:
16: sync(): void                                                    ▷ Propagates local operations to remote replicas
17:   ops = opsToPropagate()
18:   compactedOps = compact(ops)                                   ▷ Compacts the set of operations
19:   mcast(compactedOps)
20:   log_coreLocal = {}
21:   log_local = log_local \ ops
22:   log_recv = log_recv ∪ ops
23:
24: on receive(ops): void                                           ▷ Process remote operations
25:   log_recv = log_recv ∪ ops
26:   S = S • ops
```

The algorithm maintains the state of the object and two sets of operations: log_{local}, the set of effect-update operations generated in the local replica and not yet propagated to other replicas; log_{recv}, the set of effect-update operations propagated to all replicas (including operations generated locally and remotely).

When an effect-update operation is generated, the *execOp* function is called. This function adds the new operation to the log of local operations and updates the local object state.

The function *sync* is called to propagate local operations to remote replicas. It starts by computing which new operations need to be propagated, compacts the resulting set of operations for efficiency purposes, multicasts the compacted set of operations, and finally updates the local sets of operations. When a replica receives a set of operations (line 24), the set of operations propagated to all nodes and the local object state are updated accordingly.

Function *opsToPropagate* addresses the key challenge of deciding which operations need to be propagated to other replicas. To this end, we divide the operations in four groups.

First, the *forever masked* operations, which are operations that will remain in the set of masked operations independently of the operations that might be executed in the future. In the top example, an operation that adds a pair masks forever all known operations that added a pair for the same element with a lower value. These operations are removed from the set of local operations.

Second, the *core* operations (*opsImpact*, line 12), as computed locally. These operations need to be propagated, as they will (typically) impact the observable state at every replica.

Third, the operations that might impact the observable state when considered in combination with other non-core operations that might have been executed in other replicas (*opsPotImpact*, line 13). As there is no way to know which non-core operations have been executed in other replicas, it is necessary to propagate these operations also. For example, consider a modified top object where the value associated with each element is the sum of the values of the pairs added to the object. In this case, an add operation that would not move an element to the top in a replica would be in this category because it could influence

the top when combined with other concurrent adds for the same element.

Fourth, the remaining operations that might impact the observable state in the future, depending on the evolution of the observable state. These operations remain in log_{local}. In the original top example, an operation that adds a pair that will not be in the top, as computed locally, is in this category as it might become the top element after removing the elements with larger values.

For proving that the algorithm can be used to provide non-uniform eventual consistency, we need to prove the following property.

▶ **Theorem 5.** *Algorithm 1 guarantees that in a quiescent state, considering all operations O in an execution, all replicas have received all operations in a core set O_{core}.*

Proof. To prove this property, we need to prove that there exists no operation that has not been propagated by some replica and that is required for any O_{core} set. Operations in the first category have been identified as masked operations independently of any other operations that might have been or will be executed. Thus, by definition of masked operations, a O_{core} set will not (need to) include these operations. The fourth category includes operations that do not influence the observable state when considering all executed operations – if they might have impact, they would be in the third category. Thus, these operations do not need to be in a O_{core} set. All other operations are propagated to all replicas. Thus, in a quiescent state, every replica has received all operations that impact the observable state. ◀

4.4 Fault-tolerance

Non-uniform replication aims at reducing the cost of communication and the size of replicas, by avoiding propagating operations that do not influence the observable state of the object. This raises the question of the durability of operations that are not immediately propagated to all replicas.

One way to solve this problem is to have the source replica propagating every local operation to f more replicas to tolerate f faults. This ensures that an operation survives even in the case of f faults. We note that it would be necessary to adapt the proposed algorithm, so that in the case a replica receives an operation for durability reasons, it would propagate the operation to other replicas if the source replica fails. This can be achieved by considering it as any local operation (and introducing a mechanism to filter duplicate reception of operations).

4.5 Causal consistency

Causal consistency is a popular consistency model for replicated systems [15, 2, 16], in which a replica only executes an operation after executing all operations that causally precede it [12]. In the non-uniform replication model, it is impossible to strictly adhere to this definition because some operations are not propagated (immediately), which would prevent all later operations from executing.

An alternative would be to restrict the dependencies to the execution of core operations. The problem with this is that the status of an operation may change by the execution of another operation. When a non-core operation becomes core, a number of dependencies that should have been enforced might have been missed in some replicas.

We argue that the main interest of causal consistency, when compared with eventual consistency, lies in the semantics provided by the object. Thus, in the designs that we present in the next section, we aim to guarantee that in a quiescent state, the state of the replicated objects provide equivalent semantics to that of a system that enforces causal consistency.

5 Non-uniform operation-based CRDTs

CRDTs [23] are data-types that can be replicated, modified concurrently without coordination and guarantee the eventual consistency of replicas given that all updates propagate to all replicas. We now present the design of two useful operation-based CRDTs [23] that adopt the non-uniform replication model. Unlike most operation-based CRDT designs, we do not assume that the system propagates operations in a causal order. These designs were inspired by the state-based computational CRDTs proposed by Navalho *et al.* [19], which also allow replicas to diverge in their quiescent state.

5.1 Top-K with removals NuCRDT

In this section we present the design of a non-uniform top-K CRDT, as the one introduced in section 3.1. The data type allows access to the top-K elements added and can be used, for example, for maintaining the leaderboard in online games. The proposed design could be adapted to define any CRDT that filters elements based on a deterministic function by replacing the *topK* function used in the algorithm by another filter function.

For defining the semantics of our data type, we start by defining the happens-before relation among operations. To this end, we start by considering the happens-before relation established among the events in the execution of the replicated system [12]. The events that are considered relevant are: the generation of an operation at the source replica, and the dispatch and reception of a message with a new operation or information that no new message exists. We say that operation op_i happens before operation op_j iff the generation of op_i happened before the generation of op_j in the partial order of events.

The semantics of the operations defined in the top-K CRDT is the following. The *add(el,val)* operation adds a new pair to the object. The *rmv(el)* operation removes any pair of *el* that was added by an operation that happened-before the *rmv* (note that this includes non-core add operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [23], where a remove has no impact on concurrent adds. The *get()* operation returns the top-K pairs in the object, as defined by the function *topK* used in the algorithm.

Algorithm 2 presents a design that implements this semantics. The prepare-update *add* operation generates an effect-update *add* that has an additional parameter consisting in a timestamp $(replicaid, val)$, with *val* a monotonically increasing integer. The prepare-update *rmv* operation generates an effect-update *rmv* that includes an additional parameter consisting in a vector clock that summarizes add operations that happened before the remove operation. To this end, the object maintains a vector clock that is updated when a new add is generated or executed locally. Additionally, this vector clock should be updated whenever a replica receives a message from a remote replica (to summarize also the adds known in the sender that have not been propagated to this replica).

Besides this vector clock, *vc*, each object replica maintains: (i) a set, *elems*, with the elements added by all *add* operations known locally (and that have not been removed yet); and (ii) a map, *removes*, that maps each element *id* to a vector clock with a summary of the add operations that happened before all removes of *id* (for simplifying the presentation of the algorithm, we assume that a key absent from the map has associated a default vector clock consisting of zeros for every replica).

The execution of an *add* consists in adding the element to the set of *elems* if the add has not happened before a previously received remove for the same element – this can happen as operations are not necessarily propagated in causal order. The execution of a *rmv* consists

Algorithm 2 Top-K NuCRDT with removals

```
1:  elems : set of ⟨id, score, ts⟩ : initial {}
2:  removes : map id ↦ vectorClock: initial []
3:  vc : vectorClock: initial []
4:
5:  GET() : set
6:    return {⟨id, score⟩ : ⟨id, score, ts⟩ ∈ topK(elems)}
7:
8:  prepare ADD(id, score)
9:    generate add(id, score, ⟨getReplicaId(), ++ vc[getReplicaId()]⟩)
10:
11: effect ADD(id, score, ts)
12:   if removes[id][ts.siteId] < ts.val then
13:     elems = elems ∪ {⟨id, score, ts⟩}
14:     vc[ts.siteId] = max(vc[ts.siteId], ts.val)
15:
16: prepare RMV(id)
17:   generate rmv(id,vc)
18:
19: effect RMV(id, vc_rmv)
20:   removes[id] = pointwiseMax(removes[id], vc_rmv)
21:   elems = elems \ {⟨id_0, score, ts⟩ ∈ elem : id = id_0 ∧ ts.val ≤ vc_rmv[ts.siteId]}
22:
23: maskedForever(log_local, S, log_recv): set of operations
24:   adds = {add(id_1, score_1, ts_1) ∈ log_local :
25:     (∃add(id_2, score_2, ts_2) ∈ log_local : id_1 = id_2 ∧ score_1 < score_2 ∧ ts_1.val < ts_2.val)∨
26:       (∃rmv(id_3, vc_rmv) ∈ (log_recv ∪ log_local) : id_1 = id_3 ∧ ts_1.val ≤ vc_rmv[ts_1.siteId]}
27:   rmvs = {rmv(id_1, vc_1) ∈ log_local : ∃rmv(id_2, vc_2) ∈ (log_local ∪ log_recv) : id_1 = id_2 ∧ vc_1 < vc_2}
28:   return adds ∪ rmvs
29:
30: mayHaveObservableImpact(log_local, S, log_recv): set of operations
31:   return {}                          ▷ This case never happens for this data type
32:
33: hasObservableImpact(log_local, S, log_recv): set of operations
34:   adds = {add(id_1, score_1, ts_1) ∈ log_local : ⟨id_1, score_1, ts_1⟩ ∈ topK(S.elems)}
35:   rmvs = {rmv(id_1, vc_1) ∈ log_local : (∃add(id_2, score_2, ts_2) ∈ (log_local ∪ log_recv) :
36:       ⟨id_2, score_2, ts_2⟩ ∈ topK(S.elems ∪ {⟨id_2, score_2, ts_2⟩}) ∧ id_1 = id_2 ∧ ts_2.val ≤ vc_1[ts_2.siteId])}
37:   return adds ∪ rmvs
38:
39: compact(ops): set of operations
40:   return ops                         ▷ This data type does not require compaction
```

in updating *removes* and deleting from *elems* the information for adds of the element that happened before the remove. To verify if an add has happened before a remove, we check if the timestamp associated with the add is reflected in the remove vector clock of the element (lines 12 and 21). This ensures the intended semantics for the CRDT, assuming that the functions used by the protocol are correct.

We now analyze the code of these functions.

Function maskedForever computes: the local adds that become masked by other local adds (those for the same element with a lower value) and removes (those for the same element that happened before the remove); the local removes that become masked by other removes (those for the same element that have a smaller vector clock). In the latter case, it is immediate that a remove with a smaller vector clock becomes irrelevant after executing the one with a larger vector clock. In the former case, a local add for an element is masked by a more recent local add for the same element but with a larger value as it is not possible to remove only the effects of the later add without removing the effect of the older one. A local add also becomes permanently masked by a local or remote remove that happened after the add.

Function mayHaveObservableImpact returns the empty set, as for having impact on

any observable state, an operation also has to have impact on the local observable state by itself.

Function hasObservableImpact computes the local operations that are relevant for computing the top-K. An add is relevant if the added value is in the top; a remove is relevant if it removes an add that would be otherwise in the top.

5.2 Top Sum NuCRDT

We now present the design of a non-uniform CRDT, Top Sum, that maintains the top-K elements added to the object, where the value of each element is the sum of the values added for the element. This data type can be used for maintaining a leaderboard in an online game where every time a player completes some challenge it is awarded some number of points, with the current score of the player being the sum of all points awarded. It could also be used for maintaining a top of the best selling products in an (online) store (or the top customers, etc).

The semantics of the operations defined in the Top Sum object is the following. The *add(id, n)* update operation increments the value associated with *id* by *n*. The *get()* read-only operation returns the top-K mappings, $id \rightarrow value$, as defined by the *topK* function (similar to the Top-K NuCRDT).

This design is challenging, as it is hard to know which operations may have impact in the observable state. For example, consider a scenario with two replicas, where the value of the last element in the top is 100. If the known score of an element is 90, an add of 5 received in one replica may have impact in the observable state if the other replica has also received an add of 5 or more. One approach would be to propagate these operations, but this would lead to propagating all operations.

To try to minimize the number of operations propagated we use the following heuristic inspired by the demarcation protocol and escrow transactions [4, 20]. For each id that does not belong to the top, we compute the difference between the smallest value in the top and the value of the id computed by operations known in every replica – this is how much must be added to the id to make it to the top: let d be this value. If the sum of local adds for the id does not exceed $\frac{d}{num.replicas}$ in any replica, the value of id when considering adds executed in all replicas is smaller than the smallest element in the top. Thus, it is not necessary to propagate add operations in this case, as they will not affect the top.

Algorithm 3 presents a design that implements this approach. The state of the object is a single variable, *state*, that maps identifiers to their current values. The only prepare-update operation, *add*, generates an effect-update *add* with the same parameters. The execution of an effect-update *add(id, n)* simply increments the value of *id* by *n*.

Function maskedForever returns the empty set, as operations in this design can never be forever masked.

Function mayHaveObservableImpact computes the set of *add* operations that can potentially have an impact on the observable state, using the approach previously explained.

Function hasObservableImpact computes the set of *add* operations that have their corresponding *id* present in the top-K. This guarantees that the values of the elements in the top are kept up-to-date, reflecting all executed operations.

Function compact takes a set of *add* operations and compacts the *add* operations that affect the same identifier into a single operation. This reduces the size of the messages sent through the network and is similar to the optimization obtained in delta-based CRDTs [1].

Algorithm 3 Top Sum NuCRDT

```
1: state : map id ↦ sum: initial []
2:
3: GET() : map
4:   return topK(state)
5:
6: prepare ADD(id, n)
7:   generate add(id, n)
8:
9: effect ADD(id, n)
10:  state[id] = state[id] + n
11:
12: MASKEDFOREVER(log_local, S, log_recv): set of operations
13:  return {}                          ▷ This case never happens for this data type
14:
15: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
16:  top = topK(S.state)
17:  adds = {add(id, _) ∈ log_local : s = sum_val({add(i, n) ∈ log_local : i = id})
18:    ∧ s ≥ ((min(sum(top)) − (S.state[id] − s)) / getNumReplicas())}
19:  return adds
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
22:  top = topK(S.state)
23:  adds = {add(id, _) ∈ log_local : id ∈ ids(top)}
24:  return adds
25:
26: COMPACT(ops): set of operations
27:  adds = {add(id, n) : id ∈ {i : add(i, _) ∈ ops}∧ n = sum({k : add(id_1, k) ∈ ops : id_1 = id})}
28:  return adds
```

5.3 Discussion

The goal of non-uniform replication is to allow replicas to store less data and use less bandwidth for replica synchronization. Although it is clear that non-uniform replication cannot be useful for all data, we believe that the number of use cases is large enough for making non-uniform replication interesting in practice. We now discuss two classes of data types that can benefit from the adoption of non-uniform replication.

The first class is that of data types for which the result of queries include only a subset of the data in the object. In this case two different situations may occur: (i) it is possible to compute locally, without additional information, if some operation is relevant (and needs to be propagated to all replicas); (ii) it is necessary to have additional information to be able to decide if some operation is relevant. The Top-K CRDT presented in section 5.1 is an example of the former. Another example includes a data type that returns a subset of the elements added based on a (modifiable) user-defined filter – e.g. in a set of books, the filter could select the books of a given genre, language, etc. The Top-Sum CRDT presented in section 5.2 is an example of the latter. Another example includes a data type that returns the 50^{th} percentile (or others) for the elements added – in this case, it is only necessary to replicate the elements in a range close to the 50^{th} percentile and replicate statistics of the elements smaller and larger than the range of replicated elements.

In all these examples, the effects of an operation that in a given moment do not influence the result of the available queries may become relevant after other operations are executed – in the Top-K with removes due to a remove of an element in the top; in the filtered set due to a change in the filter; in the Top-Sum due to a new add that makes an element relevant; and in the percentile due to the insertion of elements that make the 50^{th} percentile change. We note that if the relevance of an operation cannot change over time, the non-uniform CRDT would be similar to an optimized CRDT that discard operations that are not relevant before propagating them to other replicas.

A second class is that of data types with queries that return the result of an aggregation over the data added to the object. An example of this second class is the Histogram CRDT presented in the appendix. This data type only needs to keep a count for each element. A possible use of this data type would be for maintaining the summary of classifications given by users in an online shop. Similar approaches could be implemented for data types that return the result of other aggregation functions that can be incrementally computed [19].

A data type that supports, besides adding some information, an operation for removing that information would be more complex to implement. For example, in an Histogram CRDT that supports removing a previously added element, it would be necessary that concurrently removing the same element would not result in an incorrect aggregation result. Implementing such CRDT would require detecting and fixing these cases.

6 Evaluation

In this section we evaluate our data types that follow the non-uniform replication model. To this end, we compare our designs against state-of-the-art CRDT alternatives: delta-based CRDTs [1] that maintain full object replicas efficiently by propagating updates as deltas of the state; and computational CRDTs [19] that maintain non-uniform replicas using a state-based approach.

Our evaluation is performed by simulation, using a discrete event simulator. To show the benefit in terms of bandwidth and storage, we measure the total size of messages sent between replicas for synchronization (total payload) and the average size of replicas.

We simulate a system with 5 replicas for each object. Both our designs and the computational CRDTs support up to 2 replica faults by propagating all operations to, at least, 2 other replicas besides the source replica. We note that this limits the improvement that our approach could achieve, as it is only possible to avoid sending an operation to two of the five replicas. By either increasing the number of replicas or reducing the fault tolerance level, we could expect that our approach would perform comparatively better than the delta-based CRDTs.

6.1 Top-K with removals

We begin by comparing our Top-K design (*NuCRDT*) with a delta-based CRDT set [1] (*Delta CRDT*) and the top-K state-based computational CRDT design [19] (*CCRDT*).

The top-K was configured with K equal to 100. In each run, 500000 update operations were generated for 10000 Ids and with scores up to 250000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Given the expected usage of top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Figures 1 and 2 show the results for workloads with 5% and 0.05% of removes respectively (the other operations are adds).

In both workloads our design achieves a significantly lower bandwidth cost when compared to the alternatives. The reason for this is that our design only propagates operations that will be part of the top-K. In the delta-based CRDT, each replica propagates all new updates and not only those that are part of the top. In the computational CRDT design, every time the top is modified, the new top is propagated. Additionally, the proposed design of computational CRDTs always propagates removes.

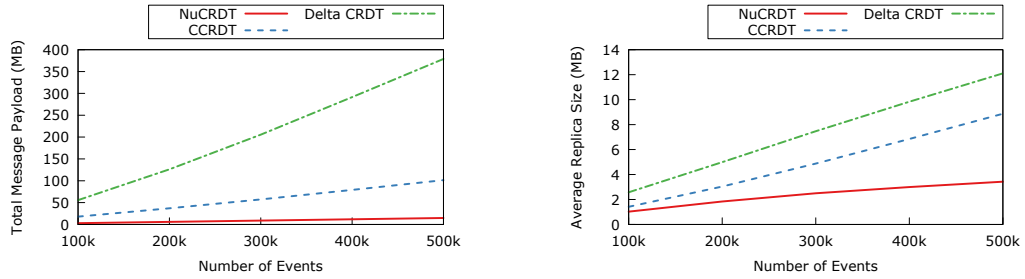


Figure 1 Top-K with removals: payload size and replica size, workload of 95/5

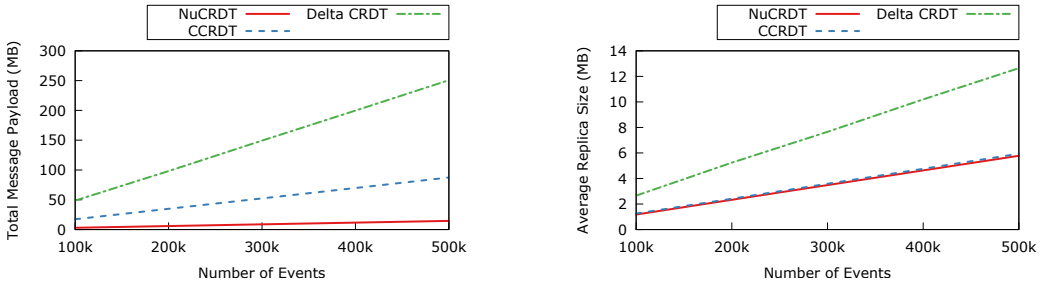


Figure 2 Top-K with removals: payload size and replica size, workload of 99.95/0.05

The results for the replica size show that our design is also more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information from remote operations received for guaranteeing fault-tolerance and those that have influenced the top-K at some moment in the execution. The computational CRDT design additionally keeps information about all removes. The delta-based CRDT keeps information about all elements that have not been removed or overwritten by a larger value. We note that as the percentage of removes approaches zero, the replica sizes of our design and that of computational CRDT starts to converge to the same value. The reason for this is that the information maintained in both designs is similar and our more efficient handling of removes starts becoming irrelevant. The opposite is also true: as the number of removes increases, our design becomes even more space efficient when compared to the computational CRDT.

6.2 Top Sum

To evaluate our Top Sum design (*NuCRDT*), we compare it against a delta-based CRDT map (*Delta CRDT*) and a state-based computational CRDT implementing the same semantics (*CCRDT*).

The top is configured to display a maximum of 100 entries. In each run, 500000 update operations were generated for 10000 Ids and with challenges awarding scores up to 1000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Figure 3 shows the results of our evaluation. Our design achieves a significantly lower bandwidth cost when compared with the computational CRDT, because in the computational CRDT design, every time the top is modified, the new top is propagated. When compared with the delta-based CRDTs, the bandwidth of NuCRDT is approximately 55% of the bandwidth used by delta-based CRDTs. As delta-based CRDTs also include a mechanism for compacting propagated updates, the improvement comes from the mechanisms for avoiding

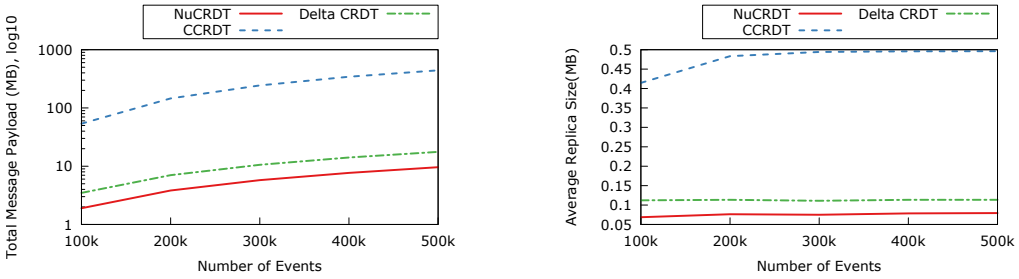


Figure 3 Top Sum: payload size and replica size

propagating operations that will not affect the top elements, resulting in less messages being sent.

The results for the replica size show that our design also manages to be more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information of remote operations received for guaranteeing fault-tolerance and those that have influenced the top elements at some moment in the execution.

7 Conclusions

In this paper we proposed the non-uniform replication model, an alternative model for replication that combines the advantages of both full replication, by allowing any replica to reply to a query, and partial replication, by requiring that each replica keeps only part of the data. We have shown how to apply this model to eventual consistency, and proposed a generic operation-based synchronization protocol for providing non-uniform replication. We further presented the designs of two useful replicated data types, the Top-K and Top Sum, that adopt this model (in appendix, we present two additional designs: Top-K without removals and Histogram). Our evaluation shows that the application of this new replication model helps to reduce the message dissemination costs and the size of replicas.

In the future we plan to study which other data types can be designed that adopt this model and to study how to integrate these data types in cloud-based databases. We also want to study how the model can be applied to strongly consistent systems.

Acknowledgments

This work has been partially funded by CMU-Portugal research project GoLocal Ref. CMUP-ERI/TIC/0046/2014, EU LightKone (grant agreement n.732505) and by FCT/MCT project NOVA-LINCS Ref. UID/CEC/04516/2013. Part of the computing resources used in this research were provided by a Microsoft Azure Research Award.

References

1 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018. doi:10.1016/j.jpdc.2017.08.003.

2 Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proc. 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013. doi:10.1145/2465351.2465361.

3 Gustavo Alonso. Partial database replication and group communication primitives. In *Proc. European Research Seminar on Advances in Distributed Systems*, 1997.

4 Daniel Barbará-Millá and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, July 1994. doi:10.1007/BF01232643.

5 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

6 Tyler Crain and Marc Shapiro. Designing a Causally Consistent Protocol for Geo-distributed Partial Replication. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015. doi:10.1145/2745947.2745953.

7 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 *Proc. 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007. doi:10.1145/1294261.1294281.

8 Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, 1987. doi:10.1145/41840.41841.

9 Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulieacute;. Epidemic Information Dissemination in Distributed Systems. *Computer*, 37(5), May 2004. doi:10.1109/MC.2004.1297243.

10 Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics. *Proc. VLDB Endow.*, 9(2):72–83, October 2015. doi:10.14778/2850578.2850582.

11 Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010. doi:10.1145/1773912.1773922.

12 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), July 1978. doi:10.1145/359545.359563.

13 Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2), May 1998. doi:10.1145/279227.279229.

14 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, 2012.

15 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 *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011. doi:10.1145/2043556.2043593.

16 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

17 Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9), July 2013. doi:10.14778/2536360.2536366.

18 Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In

Proceedings of the 26th International Conference on World Wide Web, WWW '17, pages 263–272, 2017. doi:10.1145/3038912.3052603.

19 David Navalho, Sérgio Duarte, and Nuno Preguiça. A Study of CRDTs That Do Computations. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015. doi:10.1145/2745947.2745948.

20 Patrick E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986. URL: http://doi.acm.org/10.1145/7239.7265, doi:10.1145/7239.7265.

21 Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1), March 2005. doi:10.1145/1057977.1057980.

22 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine Partial Replication in Wide Area Networks. In *Proc. 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, 2010. doi:10.1109/SRDS.2010.32.

23 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proc. 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, 2011.

24 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011. doi:10.1145/2043556.2043592.

25 Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proc. 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013. doi:10.1145/2517349.2522731.

26 Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-crdts: Making δ-crdts delta-based. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 12:1–12:4, 2016. doi:10.1145/2911151.2911163.

27 Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1), January 2009. doi:10.1145/1435417.1435432.

A APPENDIX

In this appendix we present two additional NuCRDT designs. These designs exemplify the use of different techniques for the creation of NuCRDTs.

A.1 Top-K without removals

A simpler example of a data type that fits our proposed replication model is a plain top-K, without support for the remove operation. This data type allows access to the top-K elements added to the object and can be used, for example, for maintaining a leaderboard in an online game. The top-K defines only one update operation, *add(id,score)*, which adds element *id* with score *score*. The *get()* operation simply returns the K elements with largest scores. Since the data type does not support removals, and elements added to the top-K which do not fit will simply be discarded this means the only case where operations have an impact in the observable state are if they are core operations – i.e. they are part of the top-K. This greatly simplifies the non-uniform replication model for the data type.

Algorithm 4 Top-K NuCRDT

```
1:  elems : {⟨id, score⟩} : initial {}
2:
3:  GET() : set
4:    return elems
5:
6:  prepare ADD(id, score)
7:    generate add(id, score)
8:
9:  effect ADD(id, score)
10:   elems = topK(elems ∪ {⟨id, score⟩})
11:
12: MASKEDFOREVER(log_local, S, log_recv) : set of operations
13:   adds = {add(id_1, score_1) ∈ log_local : (∃add(id_2, score_2) ∈ log_recv : id_1 = id_2 ∧ score_2 > score_1)
14:   return adds
15:
16: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
17:   return {}                                   ▷ Not required for this data type
18:
19: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
20:   return {add(id, score) ∈ log_local : ⟨id, score⟩ ∈ S.elems}
21:
22: COMPACT(ops): set of operations
23:   return ops                    ▷ This data type does not use compaction
```

Algorithm 4 presents the design of the top-K NuCRDT. The prepare-update *add(id,score)* generates an effect-update *add(id,score)*.

Each object replica maintains only a set of K tuples, *elems*, with each tuple being composed of an *id* and a *score*. The execution of *add(id,score)* inserts the element into the set, *elems*, and computes the top-K of *elems* using the function *topK*. The order used for the *topK* computation is as follows: $\langle id_1, score_1\rangle > \langle id_2, score_2\rangle$ iff $score_1 > score_2 \vee (score_1 = score_2 \wedge id_1 > id_2)$. We note that the *topK* function returns only one tuple for each element *id*.

Function MASKEDFOREVER computes the adds that become masked by other add operations for the same *id* that are larger according to the defined ordering. Due to the way the top is computed, the lower values for some given *id* will never be part of the top. Function MAYHAVEOBSERVABLEIMPACT always returns the empty set since operations in this data type are always core or forever masked. Function HASOBSERVABLEIMPACT returns the set of unpropagated add operations which add elements that are part of the top – essentially, the

add operations that are core at the time of propagation. Function COMPACT simply returns the given *ops* since the design does not require compaction.

A.2 Histogram

We now introduce the Histogram NuCRDT that maintains a histogram of values added to the object. To this end, the data type maintains a mapping of bins to integers and can be used to maintain a voting system on a website. The semantics of the operations defined in the histogram is the following: *add(n)* increments the bin n by 1; *merge(histogram$_{delta}$)* adds the information of a histogram into the local histogram; *get()* returns the current histogram.

Algorithm 5 Histogram NuCRDT

```
 1: histogram : map bin ↦ n : initial []
 2:
 3: GET() : map
 4:   return histogram
 5:
 6: prepare ADD(bin)
 7:   generate merge([bin ↦ 1])
 8:
 9: prepare MERGE(histogram)
10:   generate merge(histogram)
11:
12: effect MERGE(histogram_delta)
13:   histogram = pointwiseSum(histogram, histogram_delta)
14:
15: MASKEDFOREVER(log_local, S, log_recv) : set of operations
16:   return {}                                  ▷ Not required for this data type
17:
18: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
19:   return {}                                  ▷ Not required for this data type
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
22:   return log_local
23:
24: COMPACT(ops): set of operations
25:   deltas = {hist : merge(hist_delta) ∈ ops}
26:   return {merge(pointwiseSum(deltas))}
```

This data type is implemented in the design presented in Algorithm 5. The prepare-update *add(n)* generates an effect-update *merge([$n \mapsto 1$])*. The prepare-update operation *merge(histogram)* generates an effect-update *merge(histogram)*.

Each object replica maintains only a map, *histogram*, which maps *bins* to integers. The execution of a *merge(histogram$_{delta}$)* consists of doing a pointwise sum of the local histogram with *histogram$_{delta}$*.

Functions MASKEDFOREVER and MAYHAVEOBSERVABLEIMPACT always return the empty set since operations in this data type are always core. Function HASOBSERVABLEIMPACT simply returns log_{local}, as all operations are core in this data type. Function COMPACT takes a set of instances of *merge* operations and joins the histograms together returning a set containing only one *merge* operation.

Transparent Speculation in Geo-Replicated Transactional Data Stores

Abstract

This work presents Speculative Transaction Replication (STR), a transaction protocol for geo-distributed, partially replicated transactional data stores that exploits a form of transparent speculation to reduce inter-replica latency. In addition, we define a new consistency model, Speculative Snapshot Isolation (SPSI), that extends the semantics of Snapshot Isolation (SI) to shelter applications from the subtle anomalies that can arise from using speculative transaction processing. SPSI extends SI in an intuitive and rigorous fashion by specifying desirable atomicity and isolation guarantees that must hold when using speculative execution.

STR provides a form of speculation, called *internal* speculation, that is fully transparent for programmers (it does not expose the effects of misspeculations to clients). Since the speculation techniques employed by STR satisfy SPSI, they can be leveraged by application programs in a transparent way, with no source-code modification and no anomalous behavior. The core of STR is an innovative, fully decentralized, concurrency control mechanism, which aims not only to ensure (SPSI-)safe speculations in a lightweight and scalable fashion, but also to enhance the chances of successful speculation via a novel transaction timestamping mechanism that we called *precise clocks*.

We assess STR's performance on up to nine geo-distributed Amazon EC2 data centers, using both synthetic benchmarks as well as realistic benchmarks (TPC-C and RUBiS). Our evaluation shows that STR achieves throughput gains up to 11× and latency reduction up to 10×, in workloads characterized by low inter-data center contention. Furthermore, thanks to a self-tuning mechanism that dynamically and transparently enables and disables speculation, STR offers robust performance even when faced with unfavourable workloads that suffer from high misspeculation rates.

1. Introduction

Modern online services are increasingly deployed over geographically-scattered data centers (geo-replication) [10, 24, 26]. Geo-replication allows services to remain available even in the presence of outages affecting entire data centers and it reduces access latency by bringing data closer to clients. On the down side, though, the performance of geographically distributed data stores is challenged by large communication delays between data centers.

To provide ACID transactions, a desirable feature that can greatly simplify application development [36], some form of global (i.e., inter-data center) certification is needed to safely detect conflicts between concurrent transactions executing at different data centers. The adverse performance impact of global certification is twofold: (i) system throughput can be severely impaired, as transactions need to hold pre-commit locks during their global certification phase, which can cripple the effective concurrency that these systems can achieve; and (ii) client-perceived latency is increased, since global certification lies in the critical path of transaction execution.

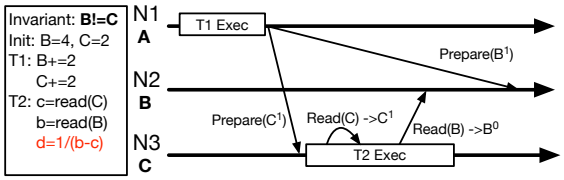
Transparent speculation. This work investigates the use of speculative processing techniques to alleviate both of the above problems. We focus on geo-distributed partially replicated transactional data stores that provide Snapshot Isolation, a widely employed consistency criterion [11, 14] (SI), and propose a novel distributed concurrency control scheme, Speculative Snapshot Isolation (SPSI), that supports a form of transparent speculative execution called *speculative reads*.

Speculative reads allow transactions to observe the data item versions produced by pre-committed transactions, instead of blocking until they are committed or aborted. As such, speculative reads can reduce the "effective duration" of pre-commit locks (i.e., as perceived by conflicting transactions), thus reducing transaction execution time and enhancing the maximum degree of parallelism achievable by the system — and, ultimately, throughput. We say that speculative reads are an *internal speculation* technique, as misspeculations caused by it never surface to the clients and can be dealt with by simply re-executing the affected transaction.

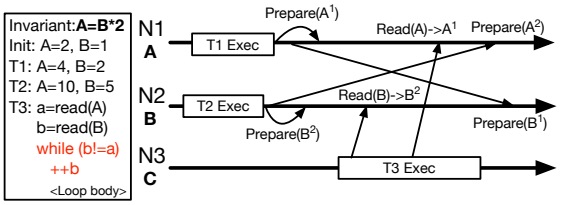
Avoiding the pitfalls of speculation. Past work has demonstrated how the use of speculation, either transparently or requiring source-code modification [16, 18, 22, 30, 39] can significantly enhance the performance of distributed [22, 30–32, 39] and single-site [16] transactional systems. However, these approaches suffer from several limitations:

1. Unfit for geo-distribution/partial replication. Some existing works in this area [22, 32, 39] were not designed for partially replicated geo-replicated data stores. On the contrary, they target

[Copyright notice will appear here once 'preprint' option is removed.]



(a) Atomicity violation — T2 observes T1's pre-committed version of data item C, but not of B. This breaks the application invariant (B≠C), causing an unexpected division by zero exception that could crash the application at node N3.



(b) Isolation violation — T3 observes the pre-committed updates of two conflicting transactions, namely T1 and T2. T3 enters an infinite loop, as the application invariant (A=B*2) is broken due to the concurrency anomaly.

Figure 1: Examples illustrating possible concurrency anomalies caused by speculative reads. N1, N2 and N3 are three nodes that store data items A, B and C, respectively.

different data models (i.e., full replication [32, 39]) or rely on techniques that impose prohibitive costs in WAN environments, such as the use of centralized sequencers to totally order transactions [22].

2. Subtle concurrency anomalies. Existing partially replicated geo-distributed transactional data stores that allow speculative reads [16, 19, 31] expose applications to anomalies that do not arise in non-speculative systems and that can severely undermine application correctness. Figure 1 illustrates two examples of concurrency anomalies that may arise with these systems. The root cause of the problem is that existing systems allow speculative reads to observe *any* pre-committed data version. This exposes applications to data snapshots that reflect only partially the updates of transactions (Fig. 1a) and/or include versions created by conflicting concurrent transactions (Fig. 1b). These anomalies have the following negative impacts: (i) transaction execution may be affected to the extent to generate anomalous/unexpected behaviours (e.g., crashing the application or hanging it in infinite loops); and (ii) they can externalize non-atomic/non-isolated snapshots to clients.

3. Performance robustness. If used injudiciously, speculation can hamper performance. As we will show, in adverse scenarios (e.g., large likelihood of transaction aborts and high system load) misspeculations can significantly penalize both user-perceived latency and system throughput.

Contributions. This paper presents the following contributions:

- Speculative Transaction Replication (STR), a novel speculative transactional protocol for partially replicated geo-distributed data stores (§5). STR shares several key design choices with state-of-the-art strongly consistent data stores [10, 11, 33], which contribute to its efficiency and scalability. These include: multi-versioning, which maximizes efficiency in read-dominated workloads [8], purely decentralized concurrency control based on loosely synchronized physical clocks [10, 11, 34], and support for partial replication [10, 23]. The key contribution of STR is its innovative, fully decentralized, concurrency control mechanism, which aims not only to ensure (SPSI-)safe speculations in a lightweight and scalable fashion, but also to enhance the chances of successful speculation via a novel transaction timestamping mechanism that we called *precise clocks*.
- Speculative Snapshot Isolation (SPSI), a novel consistency model that is the foundation of STR (§4). Besides guaranteeing the familiar Snapshot Isolation (SI) to *committed transactions*, SPSI provides clear and rigorous guarantees on the atomicity and isolation of the snapshots observed and produced by *executing transactions*. In a nutshell, SPSI requires an executing transaction to read data item versions committed before it started (as in SI), but it also allows to atomically observe the effects of non-conflicting transactions that originated on the same node and pre-committed before the transaction started. This shelters programmers from having to reason about complex concurrency anomalies that can otherwise arise in speculative systems.
- A lightweight yet effective self-tuning mechanism, based on a feedback control loop, that dynamically enables or disables speculation based on the workload characteristics (§5.5).
- We evaluate STR on up to nine geo-distributed Amazon EC2 data centers, using both synthetic and realistic benchmarks (TPC-C [4] and RUBiS [2]) (§6). Our experimental study shows that the use of internal speculation (speculative reads) yields up to 11× throughput improvements and 10× latency reduction in a fully transparent way, i.e., requiring no compensation logic.

2. Related Work

Geo-replication. The problem of designing efficient mechanisms to ensure strong consistency semantics in geo-replicated data stores has been extensively studied. A class of geo-replicated systems [12, 41] is based on the *state-machine replication* (SMR) [25] approach, in which replicas first agree on the serialization order of transactions and then execute them without further coordination. Other recent systems [10, 11, 24, 28] adopt the *deferred update* (DU) [21] approach, in which transactions are first locally executed and then globally certified. This approach is more scalable than SMR in update intensive workloads [21, 43] and, unlike SMR, it can seamlessly support non-deterministic transactions [35]. The main down side of the DU approach is that locks must be maintained for the whole duration of transactions' global certification, which can severely hinder throughput [40]. STR builds on the DU approach and tackles its performance limitation via speculative techniques.

The property introduced in this work, SPSI, is related to PSI (Parallel Snapshot Isolation) [37], a consistency criterion that relaxes SI in order to reduce latency in geo-distributed data stores. When compared with SPSI, PSI specifies a weaker consistency criterion for final committed transactions: PSI requires that transactions read the most recent committed version of some data *only if* this is created by a transaction that originated at the same site. This allows for anomalies that are not possible in SI (called long forks [37]), and that are also excluded by SPSI, which guarantees

SI-semantics for final committed transactions, i.e., they only observe the most recent committed version independently from the site in which it was originated. Further, PSI prohibits transactions from reading any version that is not final committed, which represents one of the key motivations underlying the definition of SPSI: sparing transactions from waiting for pre-commit locks to be released, while still providing rigorous consistency guarantees to shelter applications from arbitrary concurrency anomalies.

Speculation. The idea of letting transactions "optimistically" borrow, in a controlled manner, data updated by concurrent transactions has already been investigated in the past. SPECULA [32] and Aggro [29] have applied this idea to local area clusters in which data is fully replicated via total-order based coordination primitives; Jones et. al. [22] applied this idea to partially replicated/distributed databases, by relying on a central coordinator to totally order distributed transactions. These solutions provide consistency guarantees on executing transactions (and not only on committed ones) that are similar in spirit to the ones specified by SPSI. However, these systems rely on solutions (like a centralized transaction coordinator or global sequencer) that impose unacceptably large overheads in geo-distributed settings.

Other works in the distributed database literature, e.g., [16, 19, 31], have explored the idea of speculative reads (sometimes referred to as *early lock release*) in decentralized transactional protocols for partitioned databases, i.e., the same system model assumed by STR. However, these protocols provide no guarantees on the consistency of the snapshots observed by transactions (that eventually abort) during their execution and may expose applications to subtle concurrency bugs, such as the ones exemplified in Figure 1.

Another form of speculation that strives to reduce perceived-latency by exposing preliminary results to external clients, i.e., *speculative commits*, has been explored by various works. Helland et. al. advocated the *guesses and apologies* programming paradigm [20], in which systems expose preliminary results of requests (*guesses*), but reconcile the exposed results if they are different from final results (*apologies*). A similar approach is adopted also in other recent works, like PLANET [30] and ICG [18]. Unlike STR, which is totally transparent to programmers, these approaches employ a form of external speculation, which requires source-code modification to incorporate compensation logics. Furthermore, these approaches are designed to operate on conventional storage systems, which do not support speculative reads of pre-committed data. As such, although these approaches may reduce user-perceived latency, they do not tackle the problem of reducing transaction blocking time, as STR does. We will provide experimental evidence supporting this claim in§ 6.

Mixing consistency levels. Some recent systems exploit the co-existence of multiple consistency levels to enhance system performance. Gemini [26] and Indigo [6] identify and exploit the presence of commutative operations that can be executed with lightweight synchronization schemes, i.e., causal consistency, without breaking application invariants. These techniques are orthogonal to STR, which tackles the problem of enhancing the performance of non-commutative transactions that demand stronger consistency criteria (i.e., SI). Salt [44] introduced the notion of BASE transactions, i.e., classic ACID transactions that are chopped into a sequence of sub-transactions that can externalize intermediate states of their encompassing transaction to other BASE transactions. This approach, analogously to STR's speculative reads, aims to reduce lock duration and enhance throughput. Differently from STR, though, Salt requires programmers to define which intermediate states of which BASE transactions should be externalized and to reason on the correctness implications of exposing such states to other BASE transactions. STR's SPSI semantics spare programmers from this source of complexity, by ensuring that transactions always observe and produce atomic and isolated snapshots.

3. System and data model

Our target system model consists of a set of geo-distributed data centers, each hosting a set of nodes. In the following, we assume a key-value data model. This is done for simplicity and since our current implementation of STR runs on a key-value store. However, the protocol we present is agnostic to the underlying data model (e.g., relational or object-oriented).

Data and replication model. The dataset is split into multiple partitions, each of which is responsible for a disjoint key range and maintains multiple timestamped versions for each key. Partitions may be scattered across the nodes in the system using arbitrary data placement policies. Each node may host multiple partitions, but no node or data center is required to host all partitions.

A partition can be replicated within a data center and across data centers. STR employs synchronous master-slave replication to enforce fault tolerance and transparent fail over, as used, e.g., in [5, 10]. A partition has a master replica and several slave replicas. We say that a key/partition is remote for a node, if that node does not replicate that key/partition.

Synchrony assumptions. STR requires that nodes be equipped with loosely synchronized, conventional hardware clocks, which we only assume to monotonically move forward. Additional synchrony assumptions are required to ensure the correctness of the synchronous master-slave replication scheme used by STR in presence of failures [15]. STR integrates a classic single-master replication protocol, which assumes perfect failure detection capabilities [9]. However, it would be relatively straightforward to replace the replication scheme currently employed in STR to use techniques, like Paxos [13], which require weaker synchrony assumptions.

Transaction execution model. Transactions are first executed in the node where they were originated. When they request to commit, they undergo a local certification phase, which checks for conflicts with concurrent transactions in the local node. If the local certification phase succeeds, we say that transactions *local commit* and are attributed a local commit timestamp, noted LC. Next, they execute a global certification phase that detects conflicts with transactions originated at any other node in the system. Transactions that pass the global certification phase are said to *final commit* and are attributed a final commit timestamp, noted FC. Commit requests are

confirmed to applications only if the transaction is final committed, which guarantees that speculative states never surface to clients. However, the versions created by a local committed transaction T can be exposed to other transactions via the *speculative read* mechanism. We say that these transactions *data depend* on T.

4. The SPSI consistency model

We introduce Speculative Snapshot Isolation (SPSI), a consistency model that generalizes the well-known SI criterion to define a set of guarantees that shelter applications from the subtle anomalies (§Fig. 1) that may arise when using speculative techniques. Before presenting the SPSI specification, we first recall the definition of SI [42]:

- SI-1. (*Snapshot Read*) All operations read the most recent committed version as of the time when the transaction began.
- SI-2. (*No Write-Write Conflicts*) The write-sets of any committed concurrent transactions must be disjoint.

We now introduce the SPSI specification:

- SPSI-1. (*Speculative Snapshot Read*) A transaction T originated at a node N at time t must observe the most recent versions created by transactions that i) final commit with timestamp $FC \leq t$ (independently of the node where these transactions originated), and ii) local commit with timestamp $LC \leq t$ and originated at node N.
- SPSI-2. (*No Write-Write Conflicts among Final Committed Transactions*) The write-sets of any final committed concurrent transactions must be disjoint.
- SPSI-3. (*No Write-Write Conflicts among Transactions in a Speculative Snapshot*) Let S be the set of transactions included in a snapshot. The write-sets of any concurrent transactions in S must be disjoint.
- SPSI-4. (*No Dependencies from Uncommitted Transactions*) A transaction can only be final committed if it does not data depend on any local-committed or aborted transaction.

SPSI-1 extends the notion of snapshot, at the basis of the SI definition, to provide the illusion that transactions execute on immutable snapshots, which reflect the execution of all the transactions that local committed before their activation and originated on the same node. By demanding that the snapshots over which transactions execute reflect *only* the effects of locally activated transactions, SPSI allows for efficient implementations, like STR's, which can decide whether it is safe to observe the effects of a local committed transaction based solely on local information. Note that property SPSI-1 is specified for *any* transaction, including the ones that eventually abort (because some other SPSI property is violated). Hence, SPSI-1 must hold throughout the execution of transactions. This has also another relevant implication: assume that a transaction T, which started at time t, reads speculatively from a local committed transaction T' with timestamp $LC \leq t$, and that, later on, T' final commits with timestamp $FC > t$; at this point T violates the first sub-property of SPSI-1. Hence, T must be aborted before T' is allowed to final commit. The same applies in case T' aborts: since SPSI-1 prohibits developing data dependencies from aborted transactions, also in this case, T must be aborted before T' is.

SPSI-2 coincides with SI-2, ensuring the absence of write-write conflicts among concurrent final committed transactions. SPSI-3 complements SPSI-1 by ensuring that the effects of conflicting transactions can never be observed. Finally, SPSI-4 ensures that a transaction can be final committed only if it does not depend on transactions that may eventually abort.

Which anomalies does SPSI allow? SPSI provides identical guarantees to SI for final committed transactions. As for local committed and active transactions, SPSI allows for histories that would be rejected by SI, e.g., observing a version locally committed by a transaction that eventually aborts due to a conflict with some remote transaction. However, we argue that these anomalies allowed by SPSI are unharmful for applications designed to operate using SI. This is easy to show if one considers that SPSI ensures that any transaction T behaves like if it had executed under SI in a history that includes only the transactions known by the node in which T originated at the time in which T was activated. In other words, the snapshot observable by T in any SPSI-compliant history $\mathcal{H}$ is equivalent to the one that T would observe in some SI-compliant history $\mathcal{H}'$, which differs from $\mathcal{H}$ only because $\mathcal{H}'$ may omit some remote transaction concurrent with T. Clearly, if an application works correctly with SI, i.e., it is correct with any SI-compliant history (including history $\mathcal{H}'$), the application will be also be correct when faced with history $\mathcal{H}'$ — and, thus, when executing the SPSI-compliant history $\mathcal{H}$.

Which anomalies does SPSI prevent? In Fig. 1 we have already exemplified some of the concurrency anomalies that SPSI prevents, and which could lead applications to hang or crash. Interestingly, while analyzing the TPC-C and RUBiS benchmarks, we have identified several concurrency bugs that may arise and cause applications' crashes, if SPSI's guarantees are not enforced.

```
// New-Order
...
Order order;
storage->Put(order);
for (int i = 0; i < order.ol_count; i++) {
  OrderLine order_line = create_ol(order, i);
  storage->Put(order_line);
  ...
}

// Order-Status
...
Order order = storage->Read(customer.last_order);
for (int i = 0; i < order.ol_count; i++) {
  OrderLine ol = storage->Read(order.ol, i)
  //  parse throws a Null Pointer Exception if ol is null
  parse(ol);
  ...
}
```

Listing 1: Potential anomaly prevented by SPSI in TPC-C.

Listing 1 illustrates one of the anomalies we spotted in TPC-C benchmark, which involves the New Order (NO) and Order Status (OS) transactions. NO inserts a new order for a customer and then creates some number of corresponding order lines. OS fetches the identifies of the last order of a given customer, and the retrieves the corresponding order lines. In a partially-replicated setting, the order record may be stored in the node where the NO transaction

was activated, but the order lines may be stored in some different node. An injudicious use of speculative reads may allow a OS transaction to read the pre-committed order record of a concurrent NO, but then allow the OS to miss the corresponding order lines (an atomicity violation that is prevented by SPSI-1). In this case, the parse method in OS would be fed with a null pointer and generate an unexpected exception, which would never occur with SI (or SPSI) and could lead to a crash of the application.

5. The STR protocol

This section is devoted to introduce the Speculative Transaction Replication (STR) protocol. For reasons of clarity, we present the design of STR incrementally. We first present a non-speculative base protocol that implements a SI-compliant transaction system. This base protocol is then extended with a set of mechanisms aimed to support speculation in an efficient and SPSI-compliant way. Finally, we discuss the fault tolerance of STR.

Due to space constraints, we omit the presentation of the data structures used to track transaction data dependencies. We also omit the correctness proof, which will be made available after the paper is accepted.

5.1 Base non-speculative protocol

The base protocol is a multi-versioned, SI-compliant algorithm that relies on a fully decentralized concurrency control scheme similar to that employed by recent, highly scalable systems, like Spanner or Clock-SI [10, 11]. In the following, we describe the main phases of STR's base protocol.

Execution. When a transaction is activated, it is attributed a *read snapshot*, noted as *RS*, equal to the physical time of the node in which it was originated. The read snapshot determines which data item versions are visible to the transaction. Upon a read, a transaction T observes the most recent version v having final commit timestamp $v.FC \leq T.RS$. However, if there exists a pre-committed version v' with a timestamp smaller than $T.RS$, then T must wait until the pre-committed version is committed/aborted. In fact, as it will become clear shortly, the pre-committed version may eventually commit with a timestamp $FC \leq RS$ — in which case T should include it in snapshot — or $FC > RS$ — in which case it should not be visible to T.

Note that read requests can be sent to any replica that maintains the requested data item. Also, if a node receives a read request with a read snapshot RS higher than its current physical time, the node delays serving the request until its physical clock catches up with RS. Instead, writes are always processed locally and are maintained in a transaction's private buffer during the execution phase.

Certification. Read-only transactions can be immediately committed after they complete execution. Update transactions, instead, first check for write-write conflicts with *concurrent local* transactions. If T passes this local certification stage, it activates a, 2PC-based, *global certification phase* by sending a pre-commit request to the master replicas of any key it updated and for which the local node is not a master replica. If a master replica detects no conflict, it acquires pre-commit locks, and proposes its current physical time for the pre-commit timestamp.

Replication. If a master replica successfully pre-commits a transaction, it synchronously replicates the pre-commit request to its slave replicas. These, in their turn, send to the coordinator their physical time as proposed pre-commit timestamps.

Commit. After receiving replies from all the replicas of updated partitions, the coordinator calculates the commit timestamp as the maximum of the received pre-commit timestamps. Then it sends a commit message to all the replicas of updated partitions and replies to the client. Upon receiving a commit message, replicas mark the version as committed and release the pre-commit locks.

This protocol has a potential for high scalability. Unfortunately, though, in geo-distributed settings, its throughput can be severely limited by convoy effects caused by the pre-commit locks. These locks are held throughout the transactions' certification phase, which in geo-distributed data stores entail the latency of at least one inter-data center RTT —- or more if data partitions are replicated in different data-centers to allow for disaster recovery. Throughout this period, concurrent transactions attempting to read pre-committed data are conservatively blocked, which inherently limits the maximum degree of concurrency (and hence throughput) achievable by the system.

As we mentioned, the idea at the basis of STR is to tackle this problem by allowing transactions to observe pre-committed versions. Materializing this idea to build STR raised several technical challenges: guaranteeing (SPSI-)safe speculations (§ 5.2), maximizing the likelihood of successful speculation (§ 5.3) and ensuring robust performance even in adverse workload settings (§ 5.5) .

5.2 Enabling SPSI-safe speculations

Let us discuss how to extend the base protocol described above to incorporate speculative reads, while preserving SPSI semantics. The example executions in Fig. 1 illustrate two possible anomalies that could lead transactions to observe non-atomic snapshots, which violate property SPSI-1 (Fig. 1.a), or snapshots reflecting the execution of two conflicting transactions, which violate property SPSI-3 (Fig. 1.b).

STR tackles these issues as follows. First, it restricts the use of speculative reads, as mandated by SPSI-1, by allowing to observe only pre-committed versions created by local transactions. To this end, when a transaction local commits, it stores in the local node the (pre-committed) versions of the data items that it updated and that are also replicated by the local node. This is sufficient to rule out the anomalies illustrated in Fig. 1, but it still does not suffice to ensure properties SPSI-1 and SPSI-3. There are, in fact, two other subtle scenarios that have to be taken into account, both involving speculative reads of versions created by local committed transactions that updated some remote key.

The first scenario, illustrated in Fig. 2, is associated with the possibility of including in the same snapshot a local committed transaction, $T1$ — which will eventually abort due to a remote conflict, say with $T2$ — and a remote, final committed transaction, $T3$, that has read from $T2$. In fact, the totally decentralized nature

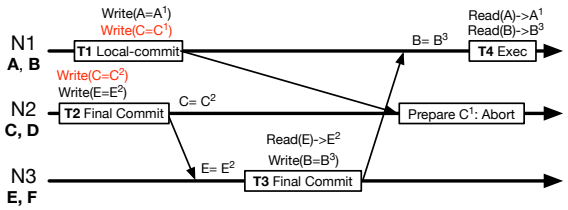


Figure 2: History exemplifying indirect conflicts between a local committed transaction, $T1$, and a final committed transaction originated at a different node, $T3$. If $T4$ included both $T1$ and $T3$ in its snapshot, it would violate SPSI property 3.

of STR's concurrency protocol, in which no node has global knowledge of all the transactions committed in the system, makes it challenging to detect scenarios like the ones illustrated in Fig. 2 and to distinguish them, in an exact way, from executions that did not include transaction $T2$ — in which case the inclusion of $T1$ and $T3$ in $T4$ would have been safe.

The mechanism that STR employs to tackle this issue is based on the observation that such scenarios can arise only in case a transaction, like $T4$, attempts to read speculatively from a local committed transaction, like $T1$, which has updated some remote key. The latter type of transactions, which we call "unsafe" transactions, may have in fact developed a remote conflict with some *concurrent* final committed transaction (which may only be detected during their global certification phase), breaking property SPSI-3. In order to detect these scenarios, STR maintains two additional data structures per transaction: OLC (Oldest Local-Commit) and FFC (Freshest Final Commit), which track, respectively, the read snapshot of the oldest "unsafe" local committed transaction and the commit timestamp of the most recent remote final committed transaction, which the current transaction has read from (either directly or indirectly). Thus, STR blocks transactions when they attempt to read versions that would cause FFC to become larger than OLC. This mechanism prevents including in the same snapshot unsafe local committed transactions along with remote final committed transactions that are concurrent and may conflict with them. For example, in Fig. 2, STR blocks $T4$ when attempting to read B from T_3, until the outcome of $T1$ is determined (not shown in the figure).

The second scenario arises in case a transaction T attempts to speculatively read a data item d that was updated by a local committed transaction T', where d is not replicated locally. In this case, if T attempted to read remotely d, it may risk to miss the version of d created by T', which would violate SPSI-1. To cope with this scenario, whenever an unsafe transaction local commits, it temporarily (until it final commits or aborts) stores the remote keys it updated in a special *cache partition*, tagging them with the same local commit timestamp. This grants prompt and atomic (i.e., all or nothing) access to these keys to any local transaction that may attempt to speculatively read them.

5.3 Promoting successful speculation via Precise Clocks

Recall that, SPSI-1 requires that if a transaction T reads speculatively from a local committed transaction T', and T' eventually final commits with a commit timestamp that is larger than the read snapshot of T, then T has to be aborted. Thus, in order to increase the chance of success of speculative reads, it is important that the commit timestamps attributed to final committed transactions are "as small as possible".

To this end, STR proposes a new timestamping mechanism, i.e., *Precise Clock*, which is based on the following observation. The smallest final commit timestamp, FC, attributable to a transaction T that has read snapshot RS must ensure the following properties:

- **P1.** $T.FC > T.RS$, which guarantees that if T reads a data item version with timestamp RS and updates it, the versions it generates has larger timestamp than the one it read.
- **P2.** $T.FC$ is larger than the read snapshot of all the transactions T_1,...,T_n, which (a) read, before T final committed, any of the keys updated by T, and (b) did not see the versions created by T, i.e., $T.FC > \max\{T_1.RS,...,T_n.RS\}$. This condition is necessary to ensure that T is serialized after the transactions T_1,...,T_n, or, in other words, to track write-after-read dependencies among transactions correctly.

Ensuring property P1 is straightforward: instead of proposing the value of the physical clock at its local node as pre-commit timestamp, the transaction coordinator proposes $T.RS + 1$. In order to ensure the latter property, STR associates to each data item an additional timestamp, called $LastReader$, which tracks the read snapshot of the most recent transaction that have read that data item. Hence, in order to ensure property P2, the nodes involved in the global certification phase of transaction T propose, as pre-commit timestamp, the maximum among the $LastReader$ timestamps of any key updated by T on that node.

It can be easily seen that the Precise Clock mechanism allows to track write-after-read dependencies among transaction at a finer granularity that the timestamping mechanism used in the base protocol — which, we recall, is also the mechanism used by non-speculative protocols like, e.g., Spanner [10] or Clock-SI [11]. Indeed, as we will show in §6, the reduction of commit timestamps, achievable via Precise Clock, does not only increase the chances of successful speculation, but also reduces abort rate for non-speculative protocols.

5.4 Algorithmic definition

Algorithms 1 and 2 give the pseudocode of the STR protocol.

Start transaction. Upon activation, a transaction is assigned a read snapshot (RS) equal to the current value of the node's physical clock. Its FFC is set to 0 and its OLCSet, i.e., the set storing the identifiers and read timestamps of the unsafe transactions from which the transaction reads from, to $<\perp,\infty>$ (Alg1, 1-6).

Speculative read. Read requests to locally-replicated keys are served by local partitions. A read request to a non-local key is first served at the cache partition to check for updates from previous local-committed transactions. If no appropriate version is found, the request is sent to any (remote) replica of the partition that contains this key (Alg1, 8-12). Upon a read request for a key, a partition updates the $LastReader$ of the key and fetches the latest version of the key with a timestamp no larger than the reader's read snapshot (Alg2, 6-7). If the fetched version is committed, or it is

Algorithm 1: Coordinator protocol

```
1  startTx()
2      Tx.RS←current_time()
3      Tx.Coord←self()
4      Tx.OLCSet← {⊥,∞}
5      Tx.FFC←0
6      return Tx

7  read(Tx, Key)
8      if Key is locally replicated or in cache then
9          {Value, Tw}← local_partition(Key).readFrom(Tx, Key)
10     else
11         send {read, Tx, Key} to any p ∈ Key.partitions()
12         wait receive {Value, Tw}
13     Tx.OLCSet.put(Tw, min_value(Tw.OLCSet)}
14     Tx.FFC←max(Tx.FFC, Tw.FFC)
15     return Value when min_value(Tx.OLCSet) >= Tx.FFC

16 commitTx(Tx)
       // Local certification
17     LCTime←Tx.RS+1
18     for P, Keys ∈ Tx.WriteSet
19         if local_replica(P).prepare(Tx) = {prepared, TS}
20             LCTime← max(LCTime, TS)
21         else
22             abort(Tx)
23     if Tx updates non-local keys
24         Tx.OLCSet.put(self(), Tx.RS)
25     send local commit requests to local replicas of updated partitions
       // Global certification
26     send prepare requests to remote master of updated partitions
27     wait receive {prepared, TS} from Tx.InvolvedReplicas
28         wait until all dependencies are resolved
29         CommitTime←max(all received TS)
30         commit(Tx, CommitTime)
31         return committed
32     wait receive aborted
33         abort(Tx)

34 commit(Tx, CT)
35     Tx.FFC←CT
36     Tx.OLCSet← {⊥,∞}
37     for Tr with data dependencies from Tx
38         if Tr.RS >= CT then
39             remove Tx from Tr's read dependency
40             Tr.OLCSet.remove(Tx)
41             Tr.FFC←max(Tr.FFC, CT)
42         else
43             abort(Tr)
44     atomically commit Tx's local committed updates
               and remove Tx's cached updates
45     send commit requests to remote replicas of updated partitions

46 abort(Tx)
47     abort transactions with dependencies from Tx
48     atomically remove Tx's local committed updates
49     send abort requests to remote replicas of updated partitions
```

local-committed and the reader is reading locally, then the partition returns the value and id of the transaction that created the value; otherwise, the reader is blocked until the transaction's final outcome is known (Alg2, 8-14). The reader transaction updates its OLCSet and FFC, and only reads the value if the minimum value in its OLCSet is greater than or equal than its FFC. If not, the transaction waits until the minimum value in its OLCSet becomes larger than its FFC (Alg1, 13-15). This condition may never become true if the transaction that created the fetched value conflicts with transactions already contained in the reader's snapshot. In that case, the reader will be aborted after this conflict is detected and stop waiting.

Local certification. After the transaction finishes execution, its write-set is locally certified. The local certification is essentially a local 2PC across all local partitions that contain keys in the transaction's write-set, including the cache partition if the transaction updated non-local keys (Alg1, 18-22). Each partition prepares the transaction if no write-write is detected, and proposes a prepare timestamp according to the Precise Clock rule (Alg2, 15-24). Upon receiving replies from all updated local partitions (including the cache partition), the coordinator calculates the local-commit timestamp as the maximum between the received prepare timestamps and the transaction's read snapshot plus one. Then, it notifies all the updated local partitions. A notified partition converts the pre-committed record to local-committed state with the local commit timestamp (Alg1, 25 and Alg2, 25-29). If the transaction updates non-local keys, the transaction is an 'unsafe' transaction, so it adds its snapshot time to its OLCSet (Alg1, 23-24).

Global certification and replication. After local certification, the keys in the transaction's write-set that have a remote master are sent to their corresponding master partitions for certification (Alg1, 26). As for the local certification phase, master partitions check for conflicts, propose a prepare timestamp and pre-commit the transaction (Alg2, 15-21). Then, a master partition replicates the prepare request to its slave replicas and replies to the coordinator (Alg2, 22-24). After receiving a replicated prepare request, the slave partition aborts any conflicting local-committed transactions and stores the prepare records. As slave replicas can be directly read bypassing their master replica, slave replicas also track the $LastReader$ for keys; so, each slave also proposes a prepare timestamp for the transaction to the coordinator (Alg2, 31-35).

Final commit/abort. A transaction coordinator can final commit a transaction, if (i) it has received prepare replies from all replicas of updated partitions, and (ii) all data dependencies and flow dependencies are resolved. The commit decision, along with the commit timestamp, is sent to to all non-local replicas of updated partitions. T's FFC is updated to its own commit timestamp, and its $OLCSet$ is set to infinity (Alg1, 35-45). Upon abort, the coordinator removes any local-committed updated version, triggers the abort of any dependent transaction and sends the decision to remote replicas (Alg1, 46-49).

5.5 Dynamically tuning speculation

Speculative reads are based on the optimistic assumption that local-committed transactions are unlikely to experience contention with remote transactions. Although our experiments in §6 show that this assumption is met in well-known benchmarks such as TPC-C and RUBiS, this is an application-dependent property. In fact, the unrestrained use of speculation in adverse workloads can lead to excessive misspeculation and degrade performance.

In order to enhance the performance robustness of STR, we coupled it with a lightweight self-tuning mechanism that dynamically decides whether to enable or disable the speculative mechanisms, depending on the workload characteristics. The tuning scheme takes a black-box approach that is agnostic of the data store implementation and also totally transparent to application developers. It relies on a simple feedback-driven control loop, steered by a centralized process that gathers measurements from all nodes in a periodic

Algorithm 2: Partition protocol

```
1  upon receiving {read, Tx, Key} by partition P
2      reply P.readFrom(Tx, Key)

3  upon receiving {prepare, Tx, Updates} by partition P
4      reply P.prepare(Tx, Updates)

5  readFrom(Tx, Key)
6      Key.LastReader←max(Key.LastReader, Tx.RS)
7      {Tw, State, Value}←KVStore.latest_before(Key, Tx.RS)
8      if State = committed
9          return{Value, Tw}
10     else if State = local-committed and local_read()
11         add data dependence from Tx to Tw
12         return{Value, Tw}
13     else
14         Tw.WaitingReaders.add(Tx)

15 prepare(Tx, Updates)
16     if exists any concurrent conflicting transaction
17         return aborted
18     else
19         PT←max(K.LastReader+1 for K ∈ Updates)
20         for{K, V} ∈ Updates do
21             KVStore.insert(K,{Tx, pre-committed, PT, V})
22         if P.isMaster() = true
23             send {replicate, Tx} to its replicas
24         return {prepared, PrepTime}

25 localCommit(Tx, LCT, Updates)
26     for{K, V} ∈ Updates do
27         KVStore.update(K,{Tx, local-committed, LCT, V})
28         unblock waiting preparing transactions
29         reply to waiting readers

30 upon receiving {replicate, Tx, Updates}
31     abort all conflicting pre-committed transactions
           and transactions read from them
32     PT←max(K.LastReader+1 for K ∈ Updates)
33     for{K, V} ∈ Updates do
34         KVStore.insert(K,{Tx, pre-committed, PT, V})
35     reply {prepared, PT}to Tx.Coord
```

fashion, compares the throughput achieved with speculative reads enabled and disabled, and accordingly configures the system.

We opted for a simple and quickly converging scheme, instead of more complex approaches (e.g., based on off-line trained classifies or more sophisticated on-line search strategies [38]), since our experimental findings confirm that, for a given workload, the decision whether or not to use speculation has a straightforward effect on throughput (no jitterlike behavior).

Our current implementation allows system administrators to initiate the self-tuning process periodically or upon request. The current self-tuning scheme could thus be naturally extended to detect statistically meaningful changes of the average input load via robust change detection algorithms, like CUSUM [7], and react to these events by re-initiating the self-tuning mechanism.

5.6 Fault tolerance

With respect to conventional/non-speculative 2PC based transactional systems, STR does not introduce additional sources of complexity for the handling of failures. Like any other approach, e.g., [10, 11, 33, 34], based on 2PC, some orthogonal mechanism (typically based on replication [17]) has to be adopted to ensure the high availability of the coordinator state.

6. Evaluation

This section presents an extensive experimental study aimed at answering the following key questions:

1. What performance gains can be achieved by STR by allowing transactions to speculatively read pre-committed data?
2. How does STR compare with systems, like PLANET [30], which employ external speculation techniques and that, unlike STR, require programmers to develop compensation logics to deal with possible misspeculations?
3. Which workloads characteristics have the strongest impact on the performance of STR?
4. How relevant is the Precise Clock technique, when used in conjunction with both speculative and non-speculative protocols?
5. How effective is STR's self-tuning mechanism to ensure robust performance in presence of workloads that are not favourable to speculative techniques?

Baselines. The first baseline protocol we consider is Clock-SI [11], which we extended to support replication, as explained in §5.1. We refer to this protocol as ClockSI-Rep. ClockSI-Rep is representative of state of the art transactional protocols based on loosely synchronized physical clocks and can be seen as the base type of systems, which we extended with speculative techniques in this work. Thus, ClockSI-Rep is an appropriate candidate to assess the performance gains achievable by using internal speculation, as well as by applying the Precise Clock technique to non-speculative systems.

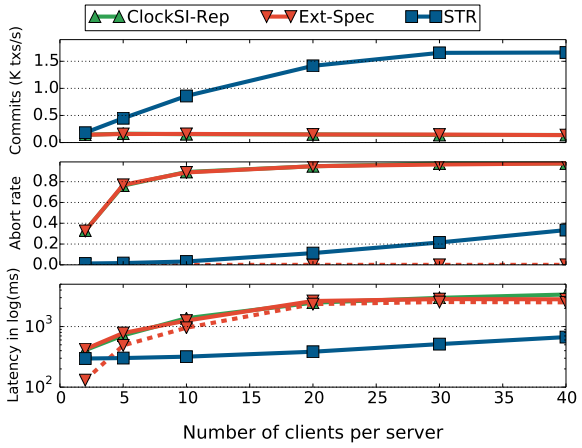
The second baseline we consider is representative of recent approaches [18, 20, 30] that propose programming models aimed to support external speculation techniques, i.e., exposing uncommitted results to clients. As discussed before, STR does not support external speculation. Support it comes at the cost of extra complexity on the programmers' side, who are forced to identify the possible concurrency anomalies that may affect their programs and develop the corresponding compensation logics. We build this baseline, which we call Ext-Spec, by developing a variant of ClockSI-Rep that externalizes to client the results of a transaction, once this passes its local certification phase and while it is still undergoing its global certification phase.

Experimental setup. We implemented the baseline protocols and STR in Erlang, based on *Antidote* [1], an open-source platform for evaluating distributed consistency protocols. The code of the protocols used in this study will be made freely available, in case this submission is accepted, in order to ensure the reproducibility of the resulted presented in the following.

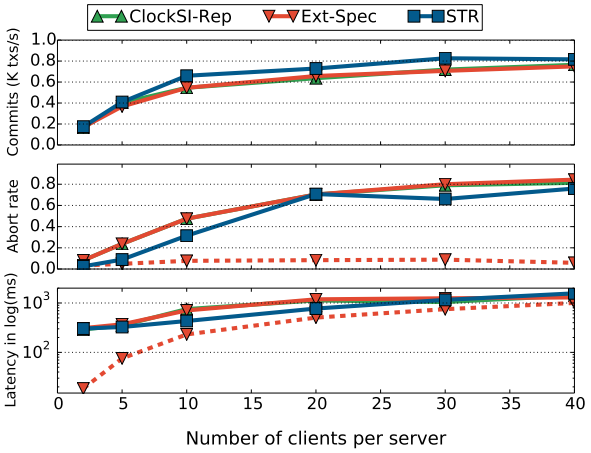
Our experimental testbed is deployed across the following nine DCs of Amazon EC2: Ireland (IE), Seoul (SU), Sydney (SY), Oregon (OR), Singapore (SG), North California (CA), Frankfurt (FR), Tokyo (TY) and North Virginia (VA). Each DC consists of three m4.large instances (2 VCPU and 8 GB of memory). We use a replication factor of six, so each partition has six replicas, and each instance holds one master replica of a partition and slave replicas of five other partitions. The above list of DCs also indicates the order of replication, e.g., a master partition located at IE has its slave replicas in SU, SY, OR, SG and CA.

	IE	SU	SY	OR	SG	CA	FR	TY	VA
Max latency (replicas)	334(SG)	267(FR)	321(FR)	160(SG)	337(IE)	167(FR)	321(SY)	212(IE)	226(SY)
Max latency (all)	334(SG)	267(FR)	321(FR)	163(SY)	337(IE)	175(SG)	324(SG)	233(FR)	226(SY)

Table 1: The first row shows the largest network latency, on average, by a DC to connect to any of its replicas; the second row shows its largest latency to all other DCs (in msec).



(a) Synth-A.



(b) Synth-B.

Figure 3: The performance of different protocols for two synthetic workloads, representative of a favourable (Synth-A) and an unfavourable (Synth-B) scenario for internal speculation. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

Load is injected by spawning one thread per emulated client in some node of the system. Each client issues transactions to a pool of local transaction coordinators and retries a transaction if it gets aborted. We use two metrics to evaluate latency: the *final latency* of a transaction is calculated as the time elapsed since its first activation until its final commit (including possible aborts and retries); for Ext-Spec, we report also the *speculative latency*, which

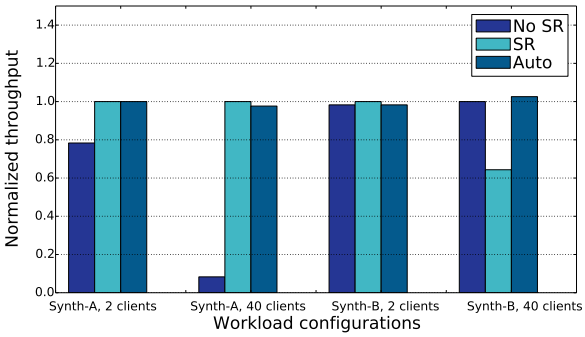


Figure 4: Normalized throughput with respect to the best performing static configuration. *No SR/SR* denote enabling/disabling statically speculative reads in STR; *Auto* denotes the use of the self-tuning technique presented in§ 5.5.

is defined as the time since the first activation of a transaction until its last speculative commit, i.e., the one after which it is final committed. Besides reporting abort rate, for Ext-Spec we also report the rate of *external misspeculation*, i.e., the percentage of transactions that were speculatively committed but finally aborted triggering the activation of some compensation logic (which we do not implement in this study, for simplicity). Each reported result is obtained from the average of at least three runs. As the standard deviations are low, we omit reporting them in the plots to enhance readability.

Unless otherwise specified, STR uses the self-tuning mechanism described in §5.5 to enable and disable the use of internal speculation. The self-tuning process gathers throughput measurements with a 10 seconds frequency. The reported results for STR refer to the final configuration identified by the self-tuning process.

6.1 Synthetic workloads

Let us start by considering a synthetic benchmark, which allows for generating workloads with precisely identifiable and very heterogeneous characteristics. The synthetic benchmark generates transactions with zero "think time", i.e., client threads issue a new transaction as soon as the previous one is final committed.

Transaction and data access. A transaction reads and updates 10 keys. When accessing a data partition, 10% of the accesses goes to a small set of keys in that data partition, which we call a hotspot, and we adjust the size of the hotspot to control contention rate. Each data partition has two million keys, of which one million are only accessible by locally-initiated transactions and the others are only accessible by remote transactions. This allows adjusting in an independent way the likelihood of contention among transactions

initiated by the same local node (local contention) and among transactions originated at remote nodes (remote contention).

We consider two workloads, which we obtain by varying the size of the hotspot sizes in the local and remote data partitions in order to synthesize two extreme scenarios that can be seen as representative of best and worst cases for internal speculation:

1. the "best case" workload, noted Synth-A, generates very high local contention, by using a single key in the hot spots of local partitions, but very low remote contention, by using 800 keys in the hot spots of remote partitions. Due to high likelihood of local contention, transactions are very likely to speculatively read versions that were local committed by some concurrent local transaction. Since remote contention is very low, though, internal speculation is very likely to succeed.
2. the "worst case" workload, noted Synth-B, has both very high local and remote contention, by using 10, resp. 3, keys in the hot spots of local, resp. remote, partitions. Like in workload Synth-A, transactions frequently use speculative reads, but, in this case, internal speculation is almost certainly doomed to fail due to the high remote contention.

Synth-A. Fig. 3.(a) clearly highlights the potential benefits that internal speculation can provide in favourable workload conditions. Both ClockSI-Rep and Ext-Spec fail to achieve any scalability and thrash, due to high abort rates (see middle plot), as soon as the degree of concurrency in the system grows to more than 2 clients. Conversely, STR scales almost linearly up to 20 clients and throughput saturates only at around 40 clients, achieving a 11.5× gain with respect to both baselines (which achieve very similar throughput levels). Also, the abort rate of STR is significantly lower than for the two baseline protocols. This is explicable considering that, with the baselines, any transaction T that read a key pre-committed by some concurrent transaction T' is forced to block; when T' commits, it is very likely that T' generates a commit timestamp larger the read snapshot of T, which causes T to abort. In the same scenario, though, STR would allow T to speculatively read from T'; also, the commit timestamp attributed to T' by Precise Clocks is likely to be smaller in absolute terms, and, with a higher probability than for the baselines, also smaller than the read timestamp of T. In this case, STR spares T from aborting, as well as from blocking — this allows STR not only to minimize the wasted work due to transactions' rollbacks, but also to enhance the degree of parallelism sustainable by the system.

It should be noted that since local contention dominates in this workload, most of the aborts occur during the local certification phase of transactions. Also, if transactions pass local certification, they are likely to avoid conflicts with remote transactions and, hence, commit with high probability. These considerations explain why Ext-Spec incurs an abort rate that is very similar to the one of ClockSI-Rep and to incur a very small external misspeculation rate.

As for the latency, the bottom plot shows about one order magnitude smaller final latency for STR compared to the baselines with more than 2 clients. This is due to the fact that both ClockSI-Rep and Ext-Spec are thrashing due to high contention in this load range. For analogous reasons, the speculative latency of Ext-Spec is only lower than the final latency of STR at very low load (2 clients), where the abort rate is still relatively low.

# of keys / **Techniques**	10	20	40	100
Physical	1/59%	1/60%	1/60%	1/72%
Precise	1.07/38%	1.07/38%	1.1/35%	1.41/48%
Physical SR	0.68/84%	0.57/83%	0.59/77%	0.97/75%
Precise SR	1.22/47%	1.21/44%	1.31/36%	1.59/49%

Table 2: Normalized throughput/abort rate of different techniques, varying a transaction's number of keys to update. *Physical/Precise* denotes the use of Physical Clock/Precise Clock; *SR* denotes that speculative reads are enabled. Throughputs reported in each column are normalized according to the throughput of 'Physical' in that column.

Synth-B. Fig. 3.(b) shows that, even in such an unfavourable workload for internal speculation, STR can provide robust performance that is at par with the baseline protocols. Thanks to its self-tuning capabilities, in fact, STR automatically disables the use of speculative reads for 30 or more clients, which correspond to load levels in which internal speculation has an adverse effect on performance.

This is illustrated in Fig. 4, which reports the performance achieved by STR with or without using speculative reads, and when using the self-tuning mechanism to select between these two configurations. More in detail, the y-axis of this figure reports the throughput of each variant of STR normalized with respect to the throughput of the variant that achieves best performance for the considered workload and number of clients (on the x-axis).

By Fig. 4, we can observe that, indeed, the use of speculative reads reduces throughput by around 40% in workload Synth-B with 40 clients and that the proposed self-tuning scheme can correctly identify the optimal configuration. By this plot, we can also observe that the choice of enabling/disabling internal speculation is not only affected by the workload type — as expected, speculative reads are beneficial in Synth-A but they are not in Synth-B — but also by the level load, fixed a given workload — speculative reads do not actually penalize throughput in Synth-B with 2 clients. Moreover, Figure 4 shows that without enabling speculative techniques, STR achieves similar throughput as the non-speculative baseline. This represents an experimental evidence supporting the efficiency of the proposed mechanism.

Benefits and overhead of Precise Clock. This experiment aims at quantifying the benefits stemming from the use of the Precise Clock mechanism, when used in conjunction with both speculative and non-speculative protocols. To this end, in Table 2, we consider four alternative systems obtained by considering ClockSI-Rep (noted *Physical*) and extending it to use Precise Clocks (noted *Precise*) and/or speculative reads (noted *SR*). In this study we vary the transactions' duration, and hence the corresponding abort cost, by varying the number of keys updated by a transaction. To maintain the contention level stable when increasing the number of keys accessed by transactions, the key space is increased by the same factor.

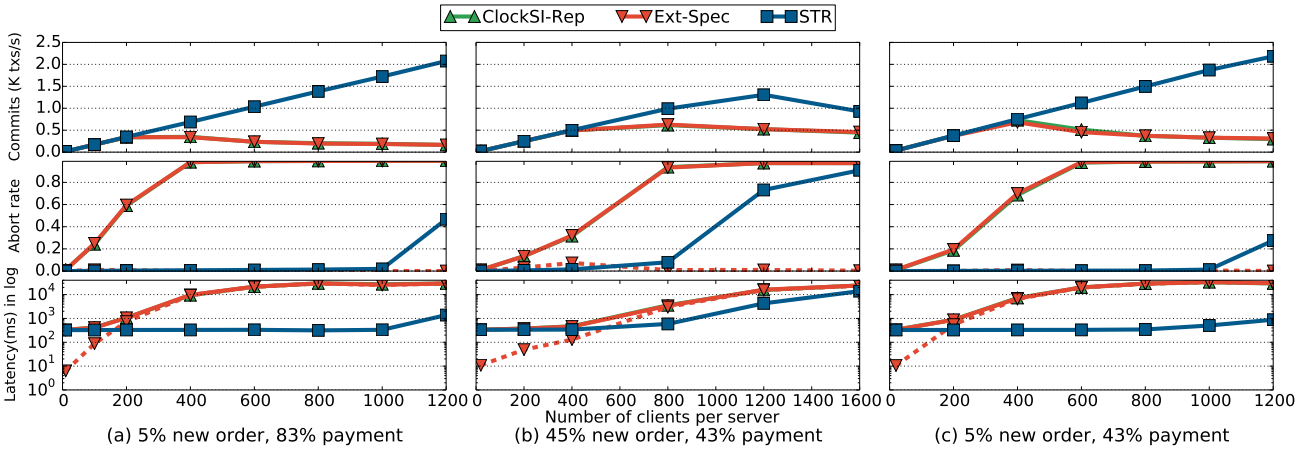


Figure 5: The performance of different protocols for three TPC-C workloads. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

Table 2 shows that Precise Clock significantly reduces abort rate and can achieve as much as 38% of throughput gain over Physical Clock for a non-speculative protocol. Generally, the more keys transactions update, the larger is the abort cost and the larger the throughput gain achieved by Precise Clock. Another interesting result is that enabling speculative reads with Physical Clock actually has negative effects on abort rate and throughput. In fact, as we have discussed in 5.3, physical clock based protocols, like Clock-SI or Spanner [10, 11], tend to generate large commit timestamp, which reduces the chances that speculative reads succeed. Finally, the collective use of Precise Clock and speculative reads results in the best throughput gain (59% for transactions updating 100 keys).

We also assessed the additional storage overhead introduced by the use of Precise Clock, which, we recall, requires maintaining additional metadata (a timestamp) for each accessed key. Our measurement shows that for the TPC-C and RUBiS benchmarks (§6.2), Precise Clock requires about 9% of extra storage.

6.2 Macro benchmarks

Next, we evaluate the performance of STR by implementing two realistic benchmarks, namely TPC-C[4] and RUBiS [2]. Unlike the previous synthetic benchmarks, TPC-C and RUBiS specify several seconds of "think time" between consecutive operations issued by a client. Hence, we need to use a much larger client population to saturate the system.

TPC-C. We implemented three TPC-C transactions, namely *payment*, *new-order* and *order-status*. The payment transaction has very high local contention and low remote contention; new-order transaction has low local contention and high remote contention, and order-status is a read-only transaction. We consider three workload mixes: 5% new-order, 83% payment and 12% order-status (TPC-C A, Fig. 5.(a)); 45% new-order, 43% payment and 12% order-status (TPC-C B, Fig. 5.(b)) and 5% new-order, 43% payment and 52% order-status (TPC-C C, Fig. 5.(c)).

Figure 5 shows that speculative reads bring significant throughput gains, as all three workloads have high degree of local contention. Compared with the baseline protocols (ClockSI-Rep and Ext-Spec), STR achieves significant speedup especially for the TPC-C A (6.13×), which has the highest degree of local contention due to having large proportion of payment transaction. For TPC-C B and TPC-C C, STR achieve 2.12× and 3× of speedup respectively. We see that the use of external speculation in this case barely brings any improvement on throughput over ClockSI-Rep. We also observe that the use of external speculation can significantly reduce the (speculative) latency perceived by clients, but only in low load conditions. This can be explained by looking at the abort rate plots, which clarly show that, as load increases, the likelihood that external speculation is successful quickly decreases.

In fact, with larger number of clients (2000 to 3000), the latency of Ext-Spec and ClockSI-Rep is on the order of 5-8 seconds, as a consequence of the high abort rate incurred by these protocols. Conversely, both Ext-Spec and STR still deliver a latency of a few hundred milliseconds.

RUBiS. RUBiS [2] models an online bidding system and encompasses 26 types of transactions, five of which are update transactions. RUBiS is designed to run on top of a SQL database, so we performed the following modifications to adapt it to STR's key-value store data model: (i) we horizontally partitioned database tables across nodes, so that each node contains an equal portion of data of each table; (ii) we created a local index for each table shard, so that some insertion operations that require a unique ID can obtain the ID locally (instead of updating a table index shared by all shards by default). We run RUBiS's 15% update default workload and use its default think time (from 2 to 10 seconds for different transactions).

Also with this benchmark (see Figure 6) STR achieves remarkable throughput gains and latency reduction. With 4000 clients (level at which we hit the memory limit and were unable to load more clients), STR achieves about 43% higher throughput. The

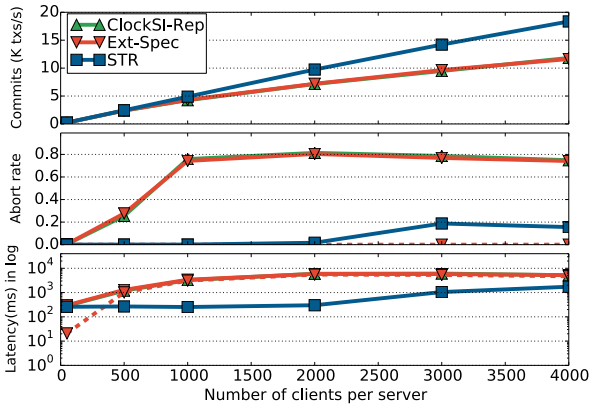


Figure 6: The performance of different protocols for RUBiS. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

final latency gains of STR over the considered baselines extends up to 10× latency reduction over ClockSI-Rep and Ext-Spec. Also in this case, external speculation is effective in reducing speculative latency only at very low load levels, before loosing effectiveness and collapsing to the same performance of ClockSI-Rep.

7. Conclusion and future work

This paper proposes STR, an innovative protocol that exploits speculative techniques to boost the performance of distributed transactions in geo-replicated settings. STR is based on a novel consistency criterion, which we call SPeculative Snapshot Isolation (SPSI). SPSI extends the familiar SI criterion and shelters programmers from subtle anomalies that can arise when adopting speculative transaction processing techniques. Furthermore, using STR requires no source-code modification, and for both of these reasons it is fully transparent to programmers.

STR builds on recent, highly scalable transactional protocols based on physical clocks (like Clock-SI and Google's Spanner) and extends them with a set of new speculative techniques (in particular, item-based timestamps to improve the speculation) and a self-tuning mechanism. Via an extensive experimental study, we show that STR can achieve striking gains (up to 11× throughput increase and 10× latency reduction) in workloads characterized by low inter-data center contention, while ensuring robust performance even in adverse settings.

We identify two main avenues for future research. The first research direction opened by this work is how to adapt both the STR protocol and its underlying speculative correctness criterion to cope with alternative consistency semantics, like Serializability or Strict Serializability. Another interesting research opportunity raised by this work is related to the design and evaluation of alternative self-tuning mechanisms, e.g., based on different modeling methodologies (e.g., relying on white-box analytical models), aimed at optimizing multiple KPIs (e.g., external mispeculation and throughput) or supporting diverse speculation degrees for different transactions' types or at different nodes in a heterogeneous cluster.

References

[1] Antidote. https://github.com/SyncFree/antidote.

[2] Rice University bidding system. http://rubis.ow2.org/.

[3] STR. https://github.com/marsleezm/STR.

[4] TPC benchmark-w specification v. 1.8. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[5] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[6] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, page 6. ACM, 2015.

[7] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[9] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[11] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184. IEEE, 2013.

[12] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone. Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 343–354. IEEE, 2014.

[13] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[14] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, pages 73–84. IEEE, 2005.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[16] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 85–96. ACM, 2013.

[17] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[18] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental consistency guarantees for replicated objects. In *12th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 16), GA, 2016. USENIX Association.

[19] J. R. Haritsa, K. Ramamritham, and R. Gupta. The prompt real-time commit protocol. *IEEE Trans. Parallel Distrib. Syst.*, 11(2):160–181, Feb. 2000.

[20] P. Helland and D. Campbell. Building on quicksand. *arXiv preprint arXiv:0909.1788*, 2009.

[21] R. Jiménez-Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 477–, Washington, DC, USA, 2002. IEEE Computer Society.

[22] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 603–614. ACM, 2010.

[23] R. Kotla, M. Balakrishnan, D. Terry, and M. K. Aguilera. Transactions with consistency choices on geo-replicated cloud storage. Technical report, September 2013.

[24] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[25] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[26] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278, 2012.

[27] Z. Li, P. Van Roy, and P. Romano. Speculative transaction processing in geo-replicated data stores. Technical Report 2, INESC-ID, Feb. 2017.

[28] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, 2013.

[29] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, pages 20–27. IEEE, 2010.

[30] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. Planet: making progress with commit processing in unpredictable environments. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 3–14. ACM, 2014.

[31] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proceedings of the VLDB Endowment*, 5(2):85–96, 2011.

[32] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. Specula: Speculative replication of software transactional memory. In *SRDS*, pages 91–100, 2012.

[33] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, pages 456–475. Springer-Verlag New York, Inc., 2012.

[34] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465. IEEE, 2012.

[35] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. PVLDB 7(10): 821-832, 2014.

[36] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[37] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[38] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[39] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, volume 29. ACM, 1995.

[40] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[41] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[42] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[43] P. T. Wojciechowski, T. Kobus, and M. Kokocinski. State-machine and deferred-update replication: Analysis and comparison. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2016.

[44] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, 2014.