Project no.	609551
Project acronym:	SyncFree
Project title:	*Large-scale computation without synchronisation*

European Seventh Framework Programme
ICT call 10

Deliverable reference number and title:	D.3.2 Protocols for divergence control and QoD
Due date of deliverable:	October 1, 2015
Actual submission date:	September 29, 2015
Start date of project:	October 1, 2013
Duration:	36 months
Name and organisation of lead editor for this deliverable:	INRIA
Revision:	0.1
Dissemination level:	CO

Contents

1 Executive Summary

The SyncFree project aims to enable trustworthy large-scale distributed applications in geo-replicated settings. The core concepts are replicated yet consistent data types (CRDTs) which allow information dissemination and sharing without the need for global synchronization.

Within the project, Work Package 3 (WP3) coordinates the work on extending the safety, quality and security guarantees provided by a system that uses minimal synchronisation. The goal of the second task, *Divergence control and quality-of-data* is to research how to address the problems that arise from divergence among replicas, i.e., the fact that at some given moment different replicas may have seen different subsets of updates and have a different state. To address these problem, this task is expected to provide Quality-of-Data (QoD) metrics such as an estimate of the amount of divergence, bound the divergence of replicas, or provide mechanisms for ensuring that global invariants are preserved in the presence of concurrent updates.

The specific requirements addressed in our work were driven mostly by the use cases studied in WP1, but also from previous experience of the project partners, both industrial and academic. We now briefly overview the results achieved during the reporting period. Some of the works are not completed as WP 3.2 runs until M30.

Quality-of-Data In a truly synchronization-free system, replicas can be modified without coordination. Thus, at any moment, the state of some replica might not reflect all the updates that have been performed in the system.

In this period, we have proposed metrics for measuring the divergence of a given replica using deterministic and probabilistic methods. We have further proposed an API that application would use for accessing the collected information. We are currently in the process of evaluating the proposed algorithms for estimating the value of each metric using a simulator developed for this specific purpose. This simulator allows to compare the value estimated with the actual value if all updates were known, thus allowing to evaluate how close the proposed algorithms are able to approach the reality.

For our simulations we are using anonymized logs of real use from Rovio and from Moodle. We expect to conclude our work until M30, when task 3.2 ends.

Invariants Although a large number of application can work correctly under weak consistency models, other applications need to maintain strong invariants that cannot be enforced using such models.

In this period we have extended our work for maintaining generic invariants expressed using any first-order logic. These extensions affected every step of the approach, namely, the static analysis for detecting operations that can break invariants and the reservation system used to enforce such invariants [4]. In a connected work, jointly with WP4, a methodology for proving that a given reservation system enforces invariants in a system was developed [14].

We have also started working on an approach to maintain invariants by repairing conflicts [5]. Our approach is based on modifying operations beforehand for guaranteeing that an invariant violation will not arise when operations are executed concurrently by leveraging the automatic resolution rules of CRDTs. This approach further reduces the need for coordination for some classes of invariants.

In this period we have also integrated our design for maintaining numeric invariants over a single object in Antidote and integrated these objects in Antidote's transactional model.

Extensions to Works Previously Reported During this period, as planned, we have also start working on security (as part of Task 3.3). We have proposed an initial algorithm for managing access control information and enforcing access control. This proposal addresses only a limited setting and will be extended in the next period.

A number of works that started being developed in the context of Task 3.1 have continued during this period, some of them leading to publications. The most relevant works, some of them being developed jointly with other Work Packages, include improvements to: (i) the model of CRDTs with delta-mutations [1] for efficient synchronization; (ii) the model of conflict-free partially replicated data types for partially caching large objects [12]; (iii) mechanisms for minimizing conflicts in transactions over partitioned data [11]; and (iv) fundamental techniques for tracking causality [13] and transferring information [26].

Basho has also been addressing the problems posed by CRDTs in terms of efficiency, as identified in the context of the Riak database. One of the ideas being explored is using delta-mutations. Some of the solutions being developed will also be used in Antidote.

2 Milestones in the Deliverable

WP3, task 2 has the following milestone, shared with other work packages:

Mil. no	Mil. name	Date due	Actual date	Lead contractor
MS1	MS2 Extended guarantees and composition in a dynamic environment	M24	M24	INRIA

Task 3.2 has contributed to this milestone (and produced research to be included in following milestones) by focusing on the following goals, as stated in the project proposal:

> This deliverable will report on the protocols for divergence control and Qo.D. This deliverable includes protocols for decentralised invariant preservation and divergence control. Synchronisation-freedom comes at the price of divergence among replicas. While many applications can work properly in this context, others require additional information, e.g., Quality-of-Data (QoD) metrics such as an estimate of the amount of divergence, or bounding the divergence, or ensuring global invariants. Compared to previous work [8, 50, 122], extreme-scale replication poses new challenges, both in the definition of divergence metrics and in the scalability of the divergence measurement and control algorithms. We will identify sub-classes of CRDTs according to the guarantees they provide, and formally analyse the degree of synchronisation that these sub-classes require. We will also explore the design space of CRDTs and associated protocols for efficiently preserving global invariants, from using decentralised solution such as escrow, reservation, and exo-leasing [74, 82], to solutions that use some synchronisation. The main challenge is to push the limits of the efficiency of the implementation of CRDTs and supporting systems for various classes of invariants, and the seamless integration of different solutions in the same platform. [month 18]

2.1 Status of Task 3.2

Task 3.2 is planned to run from M07 to M30. When making the plan of the project, we anticipated that in the context of this task, the work on Quality-of-Data (QoD) metrics would be complete before the work on invariants. Thus, we expected deliverable 3.2, due on M18, to focus primarily on QoD, with invariants being reported in deliverable 3.3 (*Protocols for invariant preservation and security*).

In the course of the project, we ended up being able to make progress on the work related with invariants more quickly than on the QoD. One of the main problems related with the work on QoD was the fact that we could only access real logs very recently – e.g. Rovio logs have been made available in July only. There has been several reasons for this delay, the most important of which the need to go through legal departments to obtain the necessary authorizations to collect and annonymize real user traces.

The delay of deliverable 3.2 to month 24 has not allowed to complete the work on QoD. In this report, we present our proposals for QoD metrics, but we have still not been able to test them against real logs. We expect that in this process, our proposals will evolve. This work will be report in the following period.

3 Contractors Contributing to the Deliverable

The following contractors contributed to the deliverables:

3.1 KL

Annette Bieniusa, Mathias Weber.

3.2 INRIA

Mahsa Najafzadeh, Jordi Martori, Marc Shapiro, Alejandro Tomsic, Tyler Crain, Pascal Urso, Marek Zawirski, Michał Jabczyński.

3.3 Louvain

Iwan Briquemont, Manuel Bravo, Zhongmiao Li, Peter van Roy.

3.4 Nova

Valter Balegas, Sérgio Duarte, Ali Shoker, Carla Ferreira, Alcino Cunha, Paulo Sérgio Almeida, Rodrigo Rodrigues, Carlos Baquero, Nuno Preguiça.

3.5 Basho

Russell Brown, Engel Sanchez, Valter Balegas (internship).

4 Results

This section presents the results obtained in WP3, during the reporting period. We organize the results in three groups: *Quality of data* (§ 4.1), discussing the work on providing information on divergence among replicas; *Invariants* (§ 4.2), detailing how to enforce invariants while minimizing the required coordination; and *Other works* (§ 4.3), describing extensions to works preciously reported.

4.1 Quality-of-Data

We now describe the work on providing information on divergence among replicas. We start by overviewing the requirements in the use cases identified in WP1 and related work. Later, we propose a set of deterministic and probabilistic divergence metrics that can be used to address the previously identified requirements (and other general cases). We finally, describe the simulation software we have developed for evaluating our proposals and a plan to integrate selected metrics in Antidote.

4.1.1 Requirements in Use Cases

Our incentive behind measuring and controlling divergence is motivated by three of the use cases offered by our industrial partners in Deliverable D.1.1: The Ad Counting Service (from Rovio), the medical Centralized National Medications and Drug Treatment System "FMK" (from Trifork), and the Music Festival App (also from Trifork). We first overview the concrete use cases, describing their relevance to the subject; and then we discuss other possible use cases.

4.1.1.1 Ad Counting Service In this service, a distributed Ad-Counter (AC) is used to count the number of times an ad is impressed. For low latency and availability, this information is maintained in multiple data centers (DCs) and servers. In general, each ad is shown in different markets or countries such that the sum of impressions should not exceed a maximum number N. This could be seen as an invariant that the system should not violate.

Since the system uses eventual-consistency (EC) to improve availability, it is possible that an ad ends up being displayed more than N times. In this application, availability and low latency is of foremost importance and exceeding N with a small number of ads n is tolerated. Thus, controlling divergence by requiring coordination among remote replicas, leading to high latency, should be avoided. Measuring the divergence and estimating the number of times an ad has been impressed can help keeping n small.

4.1.1.2 Centralized National Medications and Drug Treatment System (FMK) In FMK system, a medical profile is shared among different health institutions (hospitals, health centers, pharmacies, etc.). The problem occurs when a patient visits an institution that prescribes a drug for him and, within a short period, another institution gives him another drug that may cause some conflicts with the first one; this can take place if different institutions are not sufficiently synchronized or when failures occur.

A typical example is when a doctor prescribes a drug, the prescription should be in the pharmacy system before the patient arrive at the pharmacy to pick up the medicine. In practice, this at least takes a couple of minutes, and thus the system can be engineered accordingly. However, in the presence of faults or network partitions this no longer holds.

Therefore, it would be very useful to have an indication on how divergent the different components are for a given patient. When a doctor sees a patient's data, it is interesting to know if the latest data from, for instance, a hospital is present. In the case when data is delayed (which is frequent in this use-case) from the (unbounded) set of peripheral client-systems, ideally, it would be interesting for all relevant components to tell when they last synchronized data for a given patient. The previous measurement are deterministic metrics that can be collected from the system.

For the technical staff that maintains this system, an interesting metric would be to have some information about the status of the synchronization process with client-systems. In this case, having some aggregate measures of how many updates the client has not seen and has not propagated would be important. To this end, and to be able to provide some information during failure and partitions, it seems that it would be necessary to rely on probabilistic divergence metrics.

4.1.1.3 Music Festival App The Music Festival App is a peer-to-peer software that is used in public events and festivals where visitors can actively share their experience (comments, recommendations, like/dislike) with others. Since Internet may not be available for all visitors, this program uses all available networking mechanisms, including WiFi and Bluetooth. This requires the application to work in offline mode. The problem is that the results of segregated groups (e.g., those of Bluetooth or WiFi) are not necessarily consistent, which can give biased results.

In this application, measuring divergence is challenging since the latency among devices in offline mode is high. The reason is that people need to enable synchronization in their apps and the information needs to be propagated and collected through peer-to-peer interactions that occur spontaneously. A useful practical measurement of measurement could be number of nodes synchronized within a period of time. Relying on probabilistic metrics, in which a node estimates the divergence based on a model of how other nodes behave might also be interesting.

As for divergence control, there should be no limits to the allowable divergence in this application, but a notification or a warning is preferred if the centrally pushed information is out of date.

4.1.1.4 Summary and discussion From the above use cases, it is clear that providing deterministic divergence metrics based on time or number of operations are useful is some cases. In general, deterministic metrics, that result from measurements in (possibly multiple) nodes, can be useful for many applications and to monitor and control the system.

These metrics also serve as the basis of probabilistic divergence estimation in which a node estimates the evolution of remote nodes without contacting them. These metrics are useful to minimize communication and provide information in the presence of failures and disconnections. Although these metrics are not exact, they can help monitoring the behavior of a given system and aid in decision making.

The use cases studied do not recommend strictly controlling the divergence relying on coordination among multiple replicas, since trading off availability and increasing latency is not an option. However, it is known that a large number of applications require maintaining global invariants that require controlling the divergence among replicas. Our work on maintaining invariants is presented in Section 4.2.

We note that divergence metrics may have different uses. First, this information can be used for providing information to applications on the quality of data they are accessing. The application is then responsible to use this information for providing the best possible service to users. Second, this information can be used by users that monitor and control a replicated systems to identify potential malfunction. For example, if replica divergence deviates from historical values it is possible that there might be some malfunction due to dome failure or unexpected load. Finally, divergence information can be used internally by the system to automatically start action in an autonomic way. We are currently only focusing on providing collecting divergence information and bounding this divergence, with particular focus on guaranteeing that by bounding divergence it is possible to maintain global application invariants.

4.1.2 Related Work

A number of works previously addressed the problem of providing information about divergence and controlling this divergence. We now overview related works.

In systems that use eventual consistency, it may be interesting that some objects are more up to date than others. Also, clients may perform operations on some objects, while being disconnected from a server, which results in higher divergence among replicas. In such systems, an interesting approach is to bound the divergence between replicas, with regard to some property of the system.

There are two basic approaches in limiting divergence between replicas:

- Computing the divergence between replicas, through the use of metrics. The divergence can then be bounded, by guaranteeing that the value for each metric does not vary more than a certain threshold.
- Clients can obtain reservations that allow them to guarantee that certain conditions will hold when all updates are merged. When connected, clients ask for a reservation for a given object. They can then perform operations on the object when disconnected with the guarantee that some constraints will hold.

4.1.2.1 Metrics Systems that use this approach, rely typically on three metrics to measure divergence: one for the order of operations, one for the value of data, and one for staleness [30, 24].

- Order of updates refers to the number of updates that were not applied to a replica. The higher the value of this metric, the more updates from other replicas may be seen out of order.

- Value refers to the difference in the contents of an object between replicas, or when compared to a constant. The higher the value, the largest can be the difference from the observed value and the correct value computed after merging all unreceived updates.
- Staleness refers to the maximum time a replica may be without being refreshed with the latest value. In other words, it limits the staleness of data.

These metrics can be used only to provide information on divergence or in a system that bounds the divergence. In this case, for each metric, an application can specify the limits of divergence.

Bounding each metric must be performed with a specific algorithm. We note that bounding these metrics to 0 makes the system guarantee strong consistency, since no updates are lost, objects are always at their freshest value at all replicas, and data is never stale, so clients always get a consistent view of the system. Reversely, bounding them to infinity, grants full relaxed consistency.

4.1.2.2 Reservations Reservations are typically used in systems where clients have a need to perform operations while disconnected from the server. Reservations are (leased) locks for some operations on a given object that clients can obtain while connected to the server [20, 21]. These reservations guarantee *a priori* that the result of the operations made will be successful for the client who obtained them, avoiding the need for conflict resolution.

Several types of reservations have been proposed in literature [20, 10, 21, 27], each providing different guarantees:

- Escrow reservations provide the exclusive right to use a share of a partitionable resource represented by a numerical data item (or fragmentable object [28]).
- Value-change reservations provide the exclusive right to modify the state of an existing data item (i.e., a subset of columns in some row).
- Slot reservations provide the exclusive right to insert/remove/modify data items that satisfy a given condition.
- Value-use reservations provide the right to use a given value for some data item (despite its current value).

We now survey some systems that address the problem of Quality-of-Data.

4.1.2.3 Consistency Through Operation Restriction

The escrow transaction model [20] was first introduced for permitting updates by long-lived transactions without forbidding simultaneous access by other users to the same data items. The key idea is to split the resources associated with a data item, allowing them to be used by concurrent transactions.

This model has been later used for allowing mobile clients to execute operations during periods of disconnection [17], based on a client/server architecture. The idea of escrow transactions has been later extended in systems that use a client/server architecture [21] and peer-to-peer architecture [27].

The demarcation protocol [10] is aimed at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [19] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages.

These works are closer to the problem of invariant maintenance than simply of providing information about the Quality-of-Data.

4.1.2.4 Bounded divergence

TACT TACT is a middleware system that bounds the rate of inconsistent accesses to an underlying data store [30]. TACT mediates read/write accesses to the data store, based on consistency requirements, executing them locally if no constraints are violated, or waiting to contact other remote replicas for synchronization. TACT relies on a basic data abstraction, a conit, and on a set of metrics. Conits are used to specify consistency requirements, and the metrics are bounded in order to achieve those requirements.

Each application defines the granularity of its conits and what metrics to bound according to the applications' semantics and their consistency needs. The defined metrics are Numerical Error (NE), Order Error (OE), and Staleness. *Numerical error* limits the total weight of writes that can be applied across all replicas before being propagated to a given replica. *Order error* limits the number of tentative writes (subject to reordering) that can be outstanding at any one replica, and staleness places a real-time bound on the delay of write propagation among replicas.

Bounding all metrics to 0 guarantees strong consistency, while bounding to infinity provides optimistic replication only. Bounding NE is achieved by pushing updates to other replicas and OE by pulling updates from other replicas. The decision on what to push or pull is based only on the state of the replica they are running on. Staleness is bound using real-time vectors. Even if bounding divergence might not be interesting in many of the use cases, the defined metrics (and adaptations) can be used in our work.

Mobihoc Mobihoc is a middleware implemented to support the design of multiplayer distributed games for ad-hoc networks, and provides Vector-Field Consistency (VFC) [24]. It has a client-server architecture where the node coordinates write-locks, propagation of updates and VFC enforcement. VFC is an optimistic consistency model that allows bounded divergence of the object replicas. VFC selectively and dynamically strengthens or weakens replica consistency based on the ongoing game state. Each object has a view of the system, and for that view, it is a pivot object. The consistency degree is then stronger for objects in a certain range, and grows weaker the "further" an object is from the pivot.

In VFC, consistency degrees are defined as a vector. These vectors have three dimensions (as in TACT): time, that specifies the maximum time a replica can be without being refreshed with the latest value; sequence, that specifies the maximum number of lost replica updates; and value, which specifies the maximum relative difference between replica contents, or against a constant. This vector is the maximum divergence allowed for objects in that view.

In Mobihoc, each node keeps local replicas of all objects. The server has the primary copy of objects. Reads are done locally without locking, writes need to acquire locks to prevent loss of updates. Periodically, the server starts rounds. Updates are piggybacked in round messages, and merged at the clients.

Epsilon Serializability Epsilon Serializability (ESR) [23] is a generalization of Serializability (SR) in datatbases in which queries can view an inconsistent data item that has been updated by concurrent consistent transactions. To prevent conflicts, upper and lower bounds are set on transactions so that application constraints do not break. In particular, the approach sets a query limit for read consistency (*import-limit*) and a limit for updates (*export-limit*) and then calculates lower/upper bounds for query inconsistency based on these limits (in a similar way to escrow method). The consistency measure is the distance between current value of an item and its initial value. In the context of our work, setting limits on reads and writes means reduced availability or response time which is often not accepted nowadays. However, as we discussed in the use cases, this might be a requirement for some applications.

Wu et. al. [29] introduced a practical application to ESR in a systematic approach to convert serializable designs to ESR designs. This is done through identifying the non-SR conflicts (*extension step*), and then relaxing these conflicts in ESR using controlled inconsistency (*relaxation step*). The idea is mainly to identify the *import-limit* (i.e., the accepted Read fuzziness) and *export-limit* (i.e., accepted Write updates). The transaction that exceeds the *fuzzinees bound* is then aborted. Nevertheless, this work only addressed centralizaed databases. Pu et. al. [22] discussed how to define global constraints based on the aforementioned limits in Homogeneous and Heterogeneous databases to maintain the entire DB consistency. In Homogeneous DB, a transaction is divided into sub-transactions, each is executed on a sub-DB. The constraint here is that the global fuzziness limits (export or import), which is the sum of individual sub-transactions, must not be exceeded otherwise the transaction aborts. Heterogeneous DBs are however more challenging since every sub-database has its own consistency model where all are controlled by a global coordinator. The distributed database we address here is however homogeneous, as all replicas must follow the same consistency mode, often the causal-consistency model. On the other hand, the divergence can be measured in time delays, NB of operations, and also in fuzziness distance from the local value to the global value. In our case, the global value is often unkown and must be estimated.

Consistency rationing Consistency rationing [16] is a technique for adapting the consistency requirements of applications at runtime. The goal of the system is to reduce the total monetary cost of storage requests. The price of a particular consistency level can be measured in terms of the number of service calls needed to enforce it.

This system defines three data categories, with different consistency requirements. Category A, serializable, is the strongest and the most expensive, requiring additional services to assure data consistency. Category C provides session guarantees – read your writes – within a session, and is used if the savings compensate the expected cost of inconsistency. The B category comprises data that can be processed as C or A depending on the context.

Five policies are presented to adapt the consistency of data. The general policy is based on conflict probability, which is determined by the transactions' arrival rate. The cost of inconsistency is obtained with the probability of having conflicting updates. The time policy changes the consistency when a timestamp is reached. The fixed threshold policy, allows setting a threshold which forces the system to handle the record with strong consistency when the update exceeds that limit. This allows the invariant to be broken if there is more than one update in different servers exceeding the threshold. The demarcation policy prevents this by assigning a portion of the value to each replica and allowing the replica to update it up to that value without synchronization, like escrow techniques. If an update requires more than that portion, then the operation must be executed with strong consistency or request the portion from other replica. Finally, the dynamic policy for numeric objects adjusts the threshold according to the probability of updates exceeding it.

The system architecture is composed by clients that communicate with the application servers that run inside the cloud on top of Amazon's Elastic Computing Cloud (EC2). Application servers cache data and buffer updates before sending them to the storage system. A statistical component gathers statistics about the objects. Evaluation has shown a cost reduction and performance boost, with the dynamic policy being the most effective in terms of cost and response time.

Probabilistically Bounded Staleness Eventual Consistency has become a widely used consistency model in distributed systems, however the question remains about how eventual is eventual consistency. Probabilistic Bounded Staleness (PBS) [2] answers this question for Partial Quorums.

PBS presents three metrics: *k-staleness*, that bounds the staleness of versions returned by read quorums; *t-visibility*, that bounds the time before a committed version appears to readers; and *(k,t)-staleness*, a combination of both k and t-staleness. These metrics are probabilistic, which means that they do not guarantee that staleness is limited at some bounds, but instead provide staleness bounds with varying degrees of certainty.

PBS computes the values of the metrics for a given configuration of the system, allowing system designers to define the number of replicas in their system.

4.1.3 Divergence Estimation: Deterministic Metrics

Despite the fact the CRDTs are good at improving response time while eventually ensuring convergence of replicas, it is sometimes required to assess, measure, or control the divergence across replicas before converging as mentioned in the use-case's requirements before. Form the application perspective, it helps reducing the discrepancy between replicas to improve users experience and bound divergence. From the service perspective, it helps monitoring the system for better tuning, taking actions on overloaded replicas, and improving load balancing considering divergence measures. In this section, we discuss the deterministic metrics that help monitoring the system as well as the means to measure divergence and control it. In Section4.1.6 we discuss how these metrics could be integrated in a system, considering that keeping information for all data items is not practical and would impose an unacceptable overhead.

4.1.3.1 QoD: Time Based

Elapsed Time Since Last Sync This metrics keeps track of the time that has elapsed since the last synchronization with each replica.

We propose the following API:

```
get_etsl_sync() :  time VV of #Repl positions
get_etsl_sync(repl_id) : time
repl_id: value that identifies uniquely a replica.
    time: seconds.
    #Repl: Total number of replicas.
```

The required information is collected locally at each replica by keeping the information of when the last synchronization with each replica has been performed. This function provides information about potential staleness of the local replica with respect to remote replicas (and vice-versa).

Elapsed Time Since Last Indirect Sync This metrics keeps track of the time that a given replica has synchronized with other replicas, wither directly of indirectly.

We propose the following API:

```
get_etsl_indirect_sync() :  time VV of #Repl X #Repl positions
get_etsl_indirect_sync(repl_id) : time VV of #Repl position
repl_id: value that identifies uniquely a replica.
    time: seconds.
    #Repl: Total number of replicas.
```

This requires maintaining a matrix that is updated and propagated among replicas during the synchronization process. This information allows to estimate the potential staleness of the local replica with respect to remote replicas.

Elapsed Time Since Last Fail Attempt to Sync This metrics keeps track of the elapsed time since the last failed attempt to synchronize with a each replica.

We propose the following API:

```
get_etsl_fail_sync() : time VV of #Repl positions
get_etsl_fail_sync(repl_id) : time
    repl_id: value that identifies uniquely a replica.
    time: seconds.
    #Repl: Total number of replicas.
```

The required information is collected locally at each replica when a synchronization process fails to execute. This information can be used in the process of deciding when to start a new synchronization process.

Elapsed Time For Last Sync This metrics keeps track of how long it took for the local replica to synchronize the last time to a replica. A value is maintained for each remote replica.

We propose the following API:

```
get_etfl_sync() : time VV of #Repl positions
get_etfl_sync(repl_id) : time
    repl_id: value that identifies uniquely a replica.
    time: seconds.
    #Repl: Total number of replicas.
```

The required information is collected locally at each replica during the synchronization process. This information can be used in the process of estimating how long it will take to propagate local updates the remote replicas.

Elapsed Time Since Last Update This metric keeps track of the elapsed time since the last update executed in each replica.

We propose the following API:

```
get_etsl_update() : time VV of #Repl positions
get_etsl_update(repl_id) : time
    repl_id: value that identifies uniquely a replica.
    time: seconds.
    #Repl: Total number of replicas.
```

The required information is collected during the synchronization process by registering the time of the last update for each replica, and propagating this information among replicas. In Antidote, this information is already kept for causality tracking. This information can be used to estimate how active a given object is.

A special case of this metric is the information for the local replica. This information can be used, with metrics for last synchronization, to estimate the staleness of remote replicas of each object with respect to local updates.

Elapsed Time For Last Local Update This metrics keeps track of how long it took for the local replica to send the last update to a replica. A value is kept for each remote replica.

We propose the following API:

```
get_etfl_update() : time VV of #Repl positions
get_etfl_update(repl_id) : time
    repl_id: value that identifies uniquely a replica.
    time: seconds.
    #Repl: Total number of replicas.
```

The required information is computed locally at each replica during the synchronization process. This information provides statistical information of how long remote replica stay out-of-date with respect to local updates.

Statistics For each of the previous metrics, it would be possible to provide statistical information, including average, minimum and maximum values over a given time period.

4.1.3.2 QoD: Operation Based

# of Confirmed Operations This metrics keeps track of the number of acknowledgments received by each of the replicas to updates sent by this replica.

We propose the following API:

```
get_confirmed_operations() : #Repl x N
get_confirmed_operations(repl_id) : N
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

The required information needs to be collected by is collected locally at each replica by registering the needed information during the synchronization process. This allows to measure how divergent remote replicas are due to local updates.

# of Received Operations This metrics keeps track of the number of unique operations issued by each replica that have been received locally. This information can be used to estimate the expected rate of each replica, if its value is maintained for different time periods.

We propose the following API:

```
get_received_operation() : #Repl x N
get_received_operation(repl_id) : N
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

The required information is collected locally at each replica.

# of Operations That Are Known to be Missing This metrics keeps track of the number of known operations issued by each replica that are known to be missing.

We propose the following API:

```
get_missing_operations() : #Repl x N
get_missing_operation(repl_id) : N
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

The required information is collected locally at each replica by analyzing the (missing) dependencies of received operations.

# of Local Mutations Since Last Synchronization This metrics keeps track of the number of operations executed by the local replica, since the last synchronization, with that replica.

We propose the following API:

```
get_unsynced_operations() : #Repl x N
get_unsynced_operation(repl_id) : N
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

The required information is collected locally at each replica.

4.1.3.3 QoD: Value Based

Delta Value This metrics keeps track of the deltas produced by each replica. This information can be used to estimate the expected rate of each replica, if its value is maintained for different time periods.

We propose the following API:

```
get_received_delta() : #Repl x Val
get_received_delta(repl_id) : Val
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

The required information is collected locally at each replica.

Delta Since Last Synchronization This metrics keeps track of the delta of the local replica since the last sync, with that replica.

We propose the following API:

```
get_unsynced_delta() : #Repl x Val
get_unsynced_delta(repl_id) : Val
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

4.1.3.4 Computing Deltas for Specific CRDTs We now present different ways to compute the delta, or distance between two states, for different CRDTs.

Counters We first define the distance of an increment-only counter CRDT, as defined by Shapiro et. al.[25]. In the op-based counter,the read operation rd is defined by the eval clause which returns an integer $n \in \mathbb{N}$. The state-based increment-only counter is a bit more complex as it is composed of a map from node ids to local counters. However, since a read operations rd in the eval clause also returns the sum of all map entries, the returned value remains in $\mathbb{N}$, and hence the same distance function applies as in the op-based design. consequently, in this metric space the distance is simply defined as follows:

$$\begin{aligned} \mathsf{distance} : \mathbb{N} \times \mathbb{N} &\longrightarrow \Re \\ (x, y) &\longrightarrow |x - y| \end{aligned} \tag{1}$$

Sets Many versions of sets are defined in CRDTs, e.g., G-set, write-wins OR-Set, read-wins OR-set, etc. Despite the fact that these types have different semantics, they share the same read operation which returns the non-deleted elements in the set, lets call it elements operation. Since elements does not distinguish between the categories of the elements, a generic distance that counts the elements present in one set and not in the other can be defined as follows:

$$\begin{aligned} \mathsf{distance} : \Sigma \times \Sigma &\longrightarrow \Re \\ (x, y) &\longrightarrow \mathsf{card}\,((x \setminus y) \cup (y \setminus x)) \end{aligned} \tag{2}$$

where x and y are two set states in Σ.

Maps A multi set is a map from elements to the number of elements. Although a multi-set can be viewed as a set, however, the difference is that multi-sets differentiate between element types which can be considered in quality of data. Consequently, the distance must consider different multi-set map keys too. Now, suppose that the multi-set is composed of m different keys k and corresponding values v, we start by using the absolute distance as follows:

$$\begin{gathered}\mathsf{distance} : \mathbb{N}^m \times \mathbb{N}^m \longrightarrow \Re \\ \left((v_1^1, v_1^2, \cdots, v_1^m), (v_2^1, v_2^2, \cdots, v_2^m)\right) \longrightarrow \sum_{i=1}^{m} |v_1^i - v_2^i|\end{gathered} \tag{3}$$

Equation 3 gives the absolute change for the multi-set values which does not consider the impact of the change. To get better quality of data, we need to give higher significance to the values that caused more changes than others. Therefore, two situations that can have a greater impact on the quality of data can be considered:

(1) Vertical change: a big change that occurs on few keys quality of data.

(2) Horizontal change: a change that adds/removes new keys.

For (1), we can use the Euclidean distance that squares the individual differences and thus gives higher significance for bigger differences (a different root can be used instead of the square root, in general):

$$\begin{gathered}\mathsf{distance} : \mathbb{N}^m \times \mathbb{N}^m \longrightarrow \Re \\ \left((v_1^1, v_1^2, \cdots, v_1^m), (v_2^1, v_2^2, \cdots, v_2^m)\right) \longrightarrow \sqrt{\sum_{i=1}^{m} (v_1^i - v_2^i)^2}\end{gathered} \tag{4}$$

Regarding (2), we argue that the closer the value to zero, the higher is the impact on the m-set. We can consider this issue by introducing a new weight function to give more impact to the changes on the keys as their values become closer to zero:

$$\begin{gathered}\mathsf{weight} : \mathbb{N} \times \mathbb{N} \longrightarrow \Re \\ (v_1, v_2) \longrightarrow \frac{(v_1 - v_2)}{\min(v_1, v_2)}\end{gathered} \tag{5}$$

Now, we adjust Equation 4 accordingly to have a weighted distance:

$$\begin{gathered}\mathsf{distance} : \mathbb{N}^m \times \mathbb{N}^m \longrightarrow \Re \\ \left((v_1^1, v_1^2, \cdots, v_1^m), (v_2^1, v_2^2, \cdots, v_2^m)\right) \longrightarrow \sqrt{\sum_{i=1}^{m} w^i \times (v_1^i - v_2^i)^2}\end{gathered} \tag{6}$$

where $w^i = \mathsf{weight}(v_1^i, v_2^i)$.

Now we summarize the distance functions of different CRDTs in Table 1.

Data Type	CRDTs	State	distance	Auxiliary
Counter	GCounter PNCounter	$c \in \mathbb{N}$	$\lvert c_i - c_j \rvert$	none
Set	2PSet AWORSet RWORSet G-Set	$s \in \mathcal{P}(V)$	$\mathsf{card}\left((x \setminus y) \cup (y \setminus x)\right)$	$\mathsf{card}(s)$
Map	Multi-Set	$s = (k \hookrightarrow v) \in \mathcal{P}(V) \times \mathbb{N}$	$\sqrt{\sum_{i=1}^{m} w^i \times (v_1^i - v_2^i)^2}$	$w^i = \frac{(v_1^i - v_2^i)}{\min(v_1^i, v_2^i)}$

Table 1: A summary of CRDT quality of data distance functions.

4.1.4 Divergence Control

In the previous section we introduced a number of metrics for providing information related to divergence of replicas. Although an analysis of use cases suggests that tightly controlling divergence is not interesting in most cases, we have studied how to control divergence. In this section we discuss how to control divergence based on the distance of replica states, or fuziness. Other works, such as TACT [30], have also discussed how to bound the divergence for other types of metrics which could be adapted for use in our system.

The fuzziness distance only compares a current CRDT state with the real state that is supposed to be achieved if no more operations are executed on any node. Of course, this is impossible on a single node while concurrent operations are being executed on other nodes and, therefore, controlling reads and writes on the CRDTs is required.

Control of divergence can be can be done by imposing some constraints on the defined operations of a CRDTs. In principle, these constraints are required only for read operations, however, we show in the following that such constraints are also required for write operations to maintain liveness. An operation can import (or accept) some fuzziness as input, called imported fuzziness, bounded by an *ImpLimit*, and causes some fuzziness, called exported fuzziness that is bounded by *ExpLimit*.

4.1.4.1 Fuzziness of Reads

Since Read operations are not mutating operations in CRDTs, they do not induce fuzziness on states, and thus the *ExpLimit* is always zero. On the other hand, stale values are allowed in CRDTs and thus, read operations can import some fuzziness. Currently, systems that use CRDTs without quality control accept any fuzziness a (i.e., $ImpLimit = +\infty$). Some application however requires this fuzziness to be bounded by a finite value such that $0 < ImpLimit < +\infty$. Notice that $ImpLimit \neq 0$ otherwise the system must be in sync and stale value are not accepted.

To ensure safety, an application can accept a read value as long as the ImpLimit is not exceeded. If the system attempts to abort or suspend the read operation until synchronization occurs and hopefully leading to a lower fuzziness that is within the ImpLimit then liveness will be violated. This is clearly contradictory to the purpose of AP systems (in the CAP theory context) and is therefore impractical. The only remaining option is to limit the write operations so that a read always remains in the allowed fuzziness range. We discuss this in the next section.

4.1.4.2 Fuzziness of Writes On the other side, writes in CRDTs can import and export fuzziness. Exporting fuzziness is intuitive for writes since they are mutators by definition. However, importing fuzziness is usually not an option in classical data models, e.g., serializability or epsilon serializability [23]. The reason is that, the classical DB model does not allow for any inconsistency in the database, and thus, the entire database must be in a consistent state once a transaction commits or aborts. The subsequent transaction thus starts from a consistent state, meaning that no fuzziness is imported. In eventual consistency models, e.g., BASE or CRDTs, replicas are usually loosely coordinated aiming at better availability. Consequently, concurrent writes are always allowed, given some constraints, e.g., write operations must commute in CRDTs. This means that $ImpLimit = +\infty$ for any write operation an thus we do not discuss it any further. Notice that the ImpLimit of a write operation has no impact on the ImpLimit of read (which is the basic application constraint), to the contrary of the ExpLimit which is our focus, next.

4.1.4.3 Bounds on write operations In general, $0 < ExpLimit < +\infty$. Trivially, ExpLimit must not be zero, otherwise a write operation is no more a mutator. In addition, it must not be unlimited since this violates the ImpLimit of reads, and thus it reverts the system back to classical strong consistency models (since the system will be forced to sync). As explained above, liveness is violated if reads are executed on the premise of loose coordination. To maintain a high availability, it is necessary to slightly trade the loose coordination for better quality of reads. This can be ensured if the total fuzziness exported by concurrent writes does not exceed the ImpLimit of a read. Formally, if $ImpLimit$ is the import limit of a read operation, and m writes are concurrently being executed on different replicas, the total ExpLimit of writes is as follows:

$$\sum_{i=1}^{m} ExpLimit_i \leq ImpLimit \tag{7}$$

In principle, the individual $ExpLimit_i$ can be distributed over different replicas following a certain policy, e.g., depending on the workload of a replica, and always satisfying the constraint in Equation 7. Another option is to use a demarcation protocol [10] so that replicas can dynamically change their limits. Discussing these possibilities is orthogonal to our work, and thus we assume that the $ExpLimit$ is evenly divided among all operations on all replicas.

In many cases, the exported fuzziness of a write operation can be known in CRDTs, e.g., an increment in counters. For this reason, it makes sense to define the exported limit of a replica, rather than on an operation since multiple write operations can execute on a single replica. Now we calculate the exported limit of writes on a replica i, i.e., $RepExpLimit_i$, among n replicas, as follows:

$$RepExpLimit_i \leq \frac{ImpLimit}{n} \tag{8}$$

where ImpLimit is the imported fuzziness limit allowed in an application.

4.1.4.4 Respecting the Limits Now we discuss how the bounds defined in Equations 7 and 8 can be maintained in practice. Consider a CRDT with state s that is replicated over n replicas. Assume that s was in a consistent state on all replicas at time t^0. We require that at any instant t, a replica i retains the imported and exported fuzziness, $ImpFuzz_i$ and $ExpFuzz_i$, respectively. These values represent the distance from the current state to the real state. For simplicity, we assume that one type of read/write operations is being used, and we generalize it in later sections.

A read operation r can always be executed as long as $ImpFuzz_i^r \leq ImpLimit$. As mentioned above, this can always be maintained if the limits in Equation 7 are not violated. On the other hand, a write operation w can be executed if Equation 9 below is not violated:

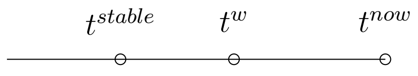
$$ExpFuzz_i \leq RepExpLimit \tag{9}$$

If this can be guaranteed once a new write is executed, the local exported fuzziness on i is updated as follows:

$$ExpFuzz_i := ExpFuzz_i + ExpFuzz_i^w \tag{10}$$

where $ExpFuzz_i^w$ is the exported fuzziness of operation w on replica i.

On the other hand, if the $RepExpLimit_i$ is reached by replica i, it must stop executing further write operations until coordinating with other replicas. Of course, it is very limited to require global synchronization, however, periodic or on-demand pair-wise coordination is possible. The basic information needed by replica i is to make sure that all other replica are aware of its current state so that it can reset its $ExpFuzz$. This information can be provided by the middleware, as in pure op-based CRDTs [8], where the middleware always retains and disseminates the last write operation w_i^{last} issued on replica i and also incorporated on all other (e.g., executed or merged), this is simply called a "stable" operation. This can be done in a periodic fashion or by a call-back to the middleware once $ExpFuzz_i$ on replica i reaches the $RepExpLimit_i$. Suppose that the last stable operation was confirmed to i at time t^{stable}, and then replica i issued a write operation w at time $t^w > t^{stable}$. By the instant that w gets confirmed to be stable, at t^{now}, replica i might have already executed some write operations, during the period $t^{now} - t^w$, as shown in the time table:



Now replica i can simply reset its $ExpFuzz_i$ as follows:

$$ExpFuzz_i := ExpFuzz_i^{now} - \mathsf{distance}(s^w, s^{stable}) \tag{11}$$

One more observation is that since the exported fuzziness of a single write w is known, then $ExpFuzz$ and $ImpFuzz$, and the limits in general, can count the number of operations instead of retaining the fuzziness, i.e., the distance.

4.1.5 Divergence Estimation: Probabilistic Metrics

The metrics introduced in the previous section return deterministic values based on value read. However, in a distributed system where replicas may evolve without coordination, these metrics provide limited information (or require a control mechanism that may restrict the execution of operations).

We now introduce probabilistic divergence metrics that provide divergence measures based on a estimated evolution of each replica. In appendix A we present a simple model to estimate data evolution. More complex models based on forecasting techniques could be used in practice (as those available in OpenForecast library [1]).

Estimated # of Missing Operations per Replica This metrics estimates the number of operations each replica has executed that have not been observed in the local replica.

We propose the following API:

```
get_estimated_operations() : #Repl x N
get_estimated_operations(repl_id) : N
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

Each replica collects information about the updates produced by each other replica. This information is collected in each replica during the normal process of synchronization, assuming the update propagation model used in Antidote. With this information, it builds a model for forecasting future evolution.

Probability of staleness of each replica This metrics estimates how probable is that a given replica has executed an update that is still not known locally.

We propose the following API:

```
get_estimated_stalenes() : #Repl x N
get_estimated_staleness(repl_id) : N
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

As before, each replica collects information about the updates produced by each other replica. With this information, it builds a model for forecasting future evolution.

Estimated delta from each replica This metrics estimates the delta each replica has produced that is still not know locally.

We propose the following API:

```
get_estimated_delta() : #Repl x Val
get_estimated_delta(repl_id) : Val
    repl_id: value that identifies uniquely a replica.
    #Repl: Total number of replicas.
```

As before, each replica collects information about the updates produced by each other replica. With this information, it builds a model for forecasting future evolution.

[1] http://www.stevengould.org/software/openforecast/index.shtml

4.1.5.1 API The previous metrics provide low-level information that may be difficult to use by application programmers. As such, we currently propose providing the following operations for accessing probabilistic divergence metrics.

Estimated Value This function returns the estimated value of a given data item.

```
get_estimated_value() : Val
```

This value can be computed from the estimated deltas for each replica.

Estimated Interval This operation returns, for a given confidence level, the interval of values for a given data item centered around the estimated value.

```
get_estimated_interval(confidenceLevel) : Interval
    confidenceLevel: value of the confidence level requested
```

The value of this function can be computed from the model that estimates the evolution of data in remote replicas.

Chance of Invariant Violation This operation returns the probability that a given operation *op* violates invariant *inv* at the current moment, assuming that the operation could be propagated immediately to other replicas. For example, in a counter, this could be used to obtain the probability that a decrement operation would lead to a violation of an invariant that specifies that the counter must be non-negative. We propose the following API:

```
invariant_violation_chance(op,inv) : Probability
    op: operation to be execued
    inv: invariant to be considered
```

The value of this function can be computed from the model that estimates the evolution of data in remote replicas.

This function could be used jointly with the invariant preservation mechanisms proposed in Section 4.2. Inn this case, whenever a local replica has not enough local right to execute an operation, if the likelihood that the operation to execute will violate the invariant, the operation could complete locally before coordinating with other replicas.

This information could also be used as a complete alternative to the invariant preservation mechanisms. For example, a client could require synchronizing with other replicas before completing an operation when the chance of invariant violation is high and proceed locally otherwise.

4.1.6 Discussion

In the previous section we have presented a set of metrics that can be used for providing information about the freshness and divergence of each data item. We have also briefly mentioned how a system could collect information for providing the value of each metric. In our work, whenever possible, we have been favoring metrics that can be computed by using local information only. The rationale for this is that it reduces the overhead for the synchronization process.

Still, maintaining the necessary information for each data item is costly. Thus, we propose that the process of maintaining divergence information is started by using an explicit call executed by clients. The information would continue to be maintained during a given time period and would end automatically if the request is not renewed. We propose the following API for this:

```
start_metric(target, aggregate, metric, duration)
stop_metric(target, aggregate, metric)
    target: target objects
    aggregate: individual or aggregate mode
    metric: metric to be collected
    duration: time for stopping collecting information
```

The parameter *aggregate* allows to specify if the information should be maintained for each data item individually or for a collection of data items individually. For example, in the medical use case discussed in Section 4.1.1, it might be important to have aggregate information about the divergence of all prescriptions instead of individual information. This can be used to identify potential problems in the replication process when the aggregated value deviates from historical data. In this case, using individual information tens to be useless, as each person typically does not have a common pattern.

4.1.7 Evaluating Divergence Algorithms

In the previous section we have presented a number of divergence metrics that can be used to provide information about the divergence a client observes when accessing the local replica.

As in a distributed system, no replica as the complete view of the system, we have decided to evaluate the relevance and quality of our divergence metrics relying on simulation and using real-life traces. In this section we describe the traces being used.

4.1.7.1 Traces In this work package we are using real-life traces in the simulations to measure system's divergence. Using real-life traces allows us to reproduce usage patterns more accurately than with random generators, since real life data accurately characterizes peaks of utilization during certain periods of the day or any other patterns specific to the domain of the service that we are studying.

We have been granted access to two sources of anonymized real-life data. The first is the trace of client interactions with Rovio's ad-service platform, a service that is similar to the Ad-service use case that was proposed in WP1. The second is the record of activity of users of a courseware platform, Moodle. The Moodle data is from the Moodle system running at NOVA, which manages information for a subset of the courses lectured at the university to the local students. We expect that both sets of data can be representative of the normal usage of real life systems and that they exhibit different patterns of users activity.

Rovio is responsible for providing the anonymized data for the Ad-service. The business logic is obfuscated and only the pattern of accesses to the underlying storage is preserved. The data from Moodle is anonymized at NOVA and hides the identitiy of the users and the contents they share.

Log Processing: We have implemented a tool to process generic logs. The platform accepts any log structured as a single operation per line, with attributes separated by a delimiter. The tool is capable of opening a number of log files and generate a stream of data to be consumed over an iterator. We support filtering the log by its attributes. This is useful, for example, to stream the list of operations generated by a subset of the users. The tool can also generate modified logs with operations ordered by some attributes. For instance, the timestamp, in the case that the multiple files of the log do not contain contiguous data. Currently, we are adding support to process compressed data, to be capable of handling the size of the Rovio logs (approximately 210GB after decompression) without consuming too much disk space.

Data Set Details:

Moodle Each log file stores all operations of the application for a single course, during a period of time. We chose a few courses that have a large user base and show activity during the whole semester to take some initial measurements. We observed that the operations are sparse over time, for which reason we are now considering the logs of multiple courses of a specific degree, or even multiple degrees to reproduce the load of the complete platform. On the initial sample of logs that we have taken (only four), 93% of operations are read only. Up to the time of writing of this report we have not analyzed the distribution of accesses over the different resources stored in the platform.

Rovio The logs that we currently have detail the number of times a given advertisement is read or updated in the storage layer. We have four full days of logs of the the system in production. However, we know that the system uses a caching layer on top of the storage that is absorbing reads and batching writes, therefore we need to know the ration of cache hits/misses or we need more detailed information to understand the real access pattern to keys. We expect to receive that information soon. We plotted the number of operations for each key, over a period of 70 minutes. Figure 4.1.7.1 shows that a only a small fraction of operations receive updates and that their popularity is similar to a power law distribution. Unfortunately, we cannot make any conclusions without understanding the effects of the caching layer.

4.1.7.2 Simulator To allow the project team to better understand the factors that affect state divergence across replicas, and to enable us to validate heuristics that enable local replicas to predict divergence values, we have designed and implemented a discrete time event-based simulator for geo-replicated systems. In the following we discuss the requirements of the simulator, followed by a discussion on the main aspects of its design and implementation.

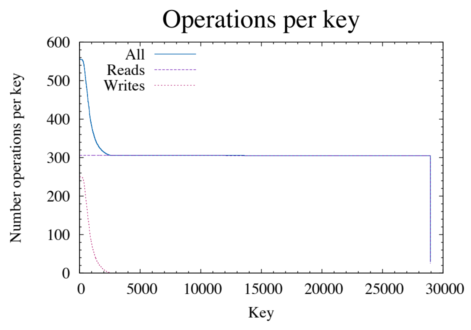


Figure 1: Number of operations per key for the Rovio traces.

Requirements: We aim at designing a simulator which has the following main properties:

Modular The simulator should be modular, namely by separating essential aspects of (complex) geo-replicated systems, such as network and communication among nodes, replication protocols employed, application logic, client logic, as well as essential aspects related with the experimental environment, such as experiment control, and reporting of results. Each of these main aspects should be materialised by modules with clearly defined interfaces that should enable the inter-operation of different implementations of these modules.

Extensible Each main module of the simulator should be extensible, enabling a user to easily extend the functionality of each of these modules and execute experiments that leverage on those extensions. For instance, the module that is responsible for materialising the network behaviour in a simulation should be easy to extend with additional behaviours such as considering message losses among nodes.

Configurable Experimental properties such as aspects or parameters related with application logic, replication strategies, and execution environment should be easy to be configured in experiments, without forcing the user to change such aspects programatically.

Model the main entities of Geo-Replicated Systems The simulator should capture the main entities of geo-replicated systems, such as data centers, servers, clients, replication protocols, applications, operations, state, and so on. The simulator should also provide fundamental mechanisms to model these entities as to simplify the design and execution of experiments by users. Having simple modulations of these entities as part of the main package of the simulator will simplify its use for fast prototyping and validation of solutions devised by the research team.

Models the Inherent Complexity of Geo-Replicated Environments Geo-Replicated environments are inherently complex, potentially being composed of servers, scattered across multiple data centers, with clients that execute operation over these different sites. Latency across machines is different data centers vary, as well as the communication latency between clients and data centers. Furthermore, network partitions that make it impossible for multiple components of the system to exchange information might occur in a transient or permanent way. Finally, components of a geo-replicated system might fail permanently (e.g, servers, clients, or data centers). All of these complexities should be captured by the simulator, enabling a user to easily trigger such conditions in their experiments and observe the effects of these conditions.

Explicitly Deals with State Divergence The main motivation for designing and implementing the simulator was to enable the research team to measure divergence between replicas under different operational loads (number of clients, workloads, environmental conditions), as well as to enable measuring the effectiveness of proposed solutions by the project consortium to address the inherent challenges of replica divergence in geo-replicated systems that leverage weak consistency. Due to this, the simulator must be able to explicitly deal and measure divergence across replicas, and expose these values to researchers executing experiments on the simulator.

Capable of Replaying Real Application Traces or Produce Artificial Traces The simulator should enable researchers to conduct experiments where geo-replicated systems are subject to operations issued by different clients that connect to different servers accordingly to real world traces, as to better capture the reality of systems. To this end, we should integrate in the simulator the ability to reply traces provided academic and industry partners of the project. Furthermore, to stress particularly challenging conditions for a system, we should also provide mechanisms that allow the simulator to generate an artificial, but controlled, load over the system.

Design and Implementation Details: We have implemented our simulator by extending and modifying the PeerSim simulator [15], a simulator which has been widely used for modelling and evaluating large-scale peer-to-peer systems written in the Java language. The original simulator supported two modes of operation (i.e, had two distinct simulation engines), one based on logical cycles, where all nodes in the system advance through the execution of synchronous computational steps (and where message exchange across different nodes is assumed to be instantaneous) and a second one based on discrete time event delivery, where simulations progress through several discrete time steps, where events can be triggered or exchanged among nodes in the system. In this simulation engine, one can easily model the communication latency by delaying the processing of communication events on the receiver node. Due to the fact that we need to capture communication latency in our experiments, as this is a key factor for divergence across replicas, we have designed our simulator by modifying the discrete time event simulation engine of the original PeerSim.

Similar to the original simulator, we model each node in a system as a stack of protocols, where each protocol is an instance of a class which provides some functionality. These modules include a *transport module*, which captures network aspects of a geo-replicated systems, such as communication latency and network partitions; a *replication module* which materializes the logic of a geo-replicated replication protocol, in particular models the exchange of information across multiple replicas; and a *application module* that captures the main logic of the application being simulated. Contrary to the original simulator, which assumed that all nodes in the simulation had the same configuration (i.e, where modeled through the same stack of protocols), we have changed the core of the simulation to allow the existence of two different type of nodes, servers and clients. While the simulator allows the user to specify the set of protocols that define each type of node, the main differences that we expect is that in most cases, client nodes will not have a replication protocol in their stack, and that the application logic for clients and servers is different (however, as discussed further ahead these aspects can be configured in the simulator through manipulation of textual configuration files).

To simplify the initial implementation of the simulator, we have modeled each data center as being composed of a single server which contains copies of all data objects. In other words, we currently assume that each data center (individually) resorts to a strong consistency replication protocols that exposes a semantic of one copy serializability, while resorting to a weak consistency replication strategy across data centers. This simplification in the implementation of the simulator will be address as future work.

We have implemented templates for fundamental strategies used in the design of (weak consistency) replication protocols whose goal is to guide users that want to model and implement their own replication protocols in the simulation to conduct experiments. These three strategies are: *i*) a reactive strategy where each data center forwards each operation received locally from clients to other data centers as soon as it processes it; *ii*) a periodic strategy where each data center aggregates operations received from clients for a configurable amount of time, and then propagate this batch of operations to the remaining data centers; and *iii*) an hybrid strategy that allows the combination of the reactive and periodic strategies in a single protocol. We not that there are many replication protocols that fall outside of these templates, and stress that the main goal of these are to guide users in the prototyping of replication protocols.

Another relevant aspect of the simulator is the support for replaying real application operation traces or generate artificial (albeit parameterised) application workloads. To this end we have modified the core of the (original) simulator to add a component which is responsible for managing the actions of clients in the system. This component has a well defined interface which is used by the simulator core to interact with it, and the concrete implementation used (which is dependent on the application being simulated) can be defined in a configuration file, without requiring any additional programatic effort from the programmer besides writing a Java class which implements the interface of this module.

4.1.7.3 Moodle Use-Case Moodle is an online course management tool also called virtual classroom. it contains different modules used to provide information and to interact with the students while off-class. There exist to main roles teachers and students. While the teacher usually does the writing operations in the course in the modules to take online exercises and message board the student also becomes a writer.

Data Structures A course has the following structures, and except for Members, there is no size bound in them. However Members has the restriction of ensuring that at least one User with the role of teacher is in the set.

Object	DT	Element	Parent	Bound
Courses	Set	Course	-	Dim(Courses) > 0
Calendars	Set	Calendar	Course	-
Assignments	Set	Assignment	Course	-
Modules	Set	Module	Course	-
Members	Set	Member	Course	$\exists$ Members.role == staff
Forums	Set	Forum	Course	-
Discussions	Set	Discussion	Forum	-
Posts	Set	Post	Discussion	-
Blogs	Set	Blog	Course	-
Directories	Set	Directory	Course	-
Files	Set	File	Directory	-
Quizes	Set	Quiz	Course	-
QuizAttempts	Set	QuizAttempt	Course	QuizAttemps.quizId $\subseteq$ Quizes
Resources	Set	Resource	Course	-
Pages	Set	Page	Course	-
Urls	Set	Url	Course	-
Users	Set	User	-	-

Operations The system provides CRUD operations (add, update, view, delete) for most data types, as well as other functionalities contained in a virtual course logic, like grading system, quiz taking, role assignation or forum subscription.

4.1.8 Future Work

In the next period we plan to complete the evaluation of the proposed divergence metrics using the developed simulator.

Additionally, we plan to implement in Antidote the most relevant divergence metrics presented in this document, or some other that might be proposed in the future. The degree of relevance will be defined according to the results obtained in our evaluation and feedback from members of the project. The implementation of the divergence metrics seems straightforward for some of the metrics (e.g. for deterministic time-based metrics, the protocols implemented for replication already propagate all the necessary information), while for others some changes to Antidote will be necessary.

We will also evalue the possibility of designing a generic system for providing divergence information relying on ESL's WOMBAT, which allows to monitor the activity in each node. Preliminary discussions with ESL seem to show that it is possible to intercept operation execution, thus allowing to obtain the necessary information for divergence without the need to modify the system being monitored.

4.2 Invariants

Systems that adopt weak consistency models have to deal with concurrent operations not seeing the effects of each other. If CRDTs can be used to guarantee eventual convergence in these cases, they cannot be used to guarantee that application invariants are enforced, which can lead to non-intuitive and undesirable semantics.

To address this problem, we continued our research on how to to maintain application invariants while minimizing coordination. The first work addresses numeric invariants, which accounts for an important class of application invariants. The second is more general and can efficiently address generic application invariants by moving coordination outside of the normal flow of operation execution. Finally, we are developing an approach that allows to enforce some invariants without any coordination, by applying the ideas of CRDTs over a set of objects that can be modified independently.

4.2.1 Enforcing Numeric Invariants

Our first work focused on enforcing numeric invariants [6] in the presence of concurrent updates to counter objects and it has been reported in the previous period. In our previous work, we showed that fast geo-replicated operations on counters can coexist with strong invariants. To this end, we proposed a novel abstract data type called Bounded Counter. This replicated object, like conventional CRDTs, allows for operations to execute locally, automatically merges concurrent updates, and, in contrast to previous counter CRDTs, also enforces numeric invariants while avoiding any coordination in most cases. This work has been accepted for publication recently [6].

In this period, we integrated the Bounded Counter in Antidote. Unlike our initial work, where each Bounded Counter is accessed and modified independently, in Antidote, a Bounded Counter can be accessed in the context of a transaction. This allows to extend the transaction model of Antidote, Transactional Causal+ Consistency, to provide much strong guarantees by enforcing invariant in some objects. Additionally, by relying on transactions that can fail in the site where they are submitted, this allows to combine multiple Bounded Counters to provide guarantees that was difficult to provide previously – e.g. to define multiple invariants over the same data item. We are currently in the process of formalizing this transactional model, which we expect to complete soon.

4.2.2 Explicit Consistency

Our second work proposes a general approach for maintaining applications invariants, based on *explicit consistency* [4]. Again, in the previous report we had already reported an initial version of this work.

Explicit consistency is a novel consistency semantics for replicated systems. The high level idea is to let programmers define the application-specific correctness rules that should be met at all times. These rules are defined as invariants over the database state.

Given the invariants expressed by the programmer, we propose a methodology for enforcing explicit consistency that has three steps: (i) detect the sets of operations that may lead to invariant violation when executed concurrently (we call these sets *I-offender sets*); (ii) select an efficient mechanism for handling *I-offender sets*; (iii) instrument the application code to use the selected mechanism in a weakly consistent database system.

In this period, we have made several improvements to the language used to specify invariants, to the static analysis process used to infer the sets of operations, *I-offender sets*, that may lead to an invariant violation and to the reservation system used to enforce invariants. These improvements are described in more detail in Balegas et. al. [4]. In this report, we only describe the complete set of reservations included in our proposal.

To illustrate the concepts, we use the example of an application that manages tournaments in a distributed multi-player game. The game maintains information about players and tournaments. Players can register and de-register from the game. Players compete in tournaments, for which they can enroll and disenroll. A set of matches occurs for each tournament. Each tournament has a maximum capacity. In some cases, e.g., when there are not enough participants, a tournament can be canceled before it starts. Otherwise a tournament's life cycle is creation, start, and end.

4.2.2.1 Reservations We have also made important extensions to the techniques used to avoid the execution of operations that can lead to invariant violation. Our reservation system is now comprised of the following techniques.

UID generator: A very common invariant is uniqueness of identifiers [18]. This problem can be easily solved, without coordination, by statically splitting the space of identifiers per replica. Indigo provides this service by appending a replica-specific suffix to a locally-unique identifier.

Multi-level lock reservation: The multi-level lock reservation (or simply multi-level lock) is our base mechanism to restrict the concurrent execution of operations that can break invariants. A multi-level lock can provide the following rights: *(i) shared forbid*, giving the shared right to forbid some action to occur; *(ii) shared allow*, giving the shared right to allow some action to occur; *(iii) exclusive allow*, giving the exclusive right to execute some action.

When a replica holds one of the above rights, no other replica holds rights of a different type. For instance, if a replica holds a *shared forbid*, no other replica has any form of *allow*. We now show how to use this knowledge to control the execution of *I-offender sets*.

In the tournament example, $\{enrollTournament(P, T), removePlayer(P)\}$ is an *I-offender set*. To avoid the violation of invariants, we can associate an appropriate multi-level lock to each of the operations, for specific values of the parameters. For example, we can have a multi-level lock associated with $removePlayer(P)$, for each value of P. For executing $removePlayer(P)$, it is necessary to obtain the right *shared allow* on the reservation for $removePlayer(P)$. For executing $enrollTournament(P, T)$, it is necessary to obtain the *shared forbid* right on the reservation for $removePlayer(P)$. This guarantees that enrolling some player will not execute concurrently with deleting the same player. However, concurrent enrolls or concurrent removes are allowed. In particular, if all replicas hold the *shared forbid* right on removing players, the most frequent enroll operation can execute in any replica, without coordination with other replicas.

The *exclusive allow* right, in turn, is necessary when an operation is incompatible with itself, i.e., when executing concurrently the same operation may lead to an invariant violation.

Multi-level locks are a form of lock that can be used to restrict the concurrent execution of operations in any *I-offender sets*. It would be possible to enforce any application invariants using only multi-level locks. However, in some cases it is possible to provide additional concurrency while enforcing invariants, by using the following reservations.

Multi-level mask reservation: For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates false may lead to an invariant violation if all other predicates were originally false. In our analysis, each of these pairs is an *I-offender set*.

Using simple multi-level locks for every pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of all operations that could make any of the other predicates false. The reason why this is overly pessimistic is that, in this case, for executing an operation that makes some predicate false it suffices to guarantee that some other predicate remains true, which can be done by only forbidding the operations that make it false.

To allow for this, Indigo includes a multi-level mask reservation that can be seen as a vector of multi-level locks. For the invariant $P_1 \vee P_2 \vee \ldots \vee P_n$, a multi-level mask with n entries is created, with entry i used to control operations that may make P_i false.

When a replica obtains a *shared allow* right in one entry, it must obtain a *shared forbid* right in some other entry. For example, an operation that may make P_i false needs to obtain the *shared allow* right on the i^{th} entry and a *shared forbid* right on an entry j for which the predicate is true. At runtime, to find an entry to forbid, it is only necessary to evaluate the current value of the predicate associated with each entry that can be locked.

Escrow reservation: For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing some decrements to execute without coordination [20]. Given an initial value for $x = x_0$, there are initially $x_0 - k$ rights to execute decrements. These rights can be split dynamically among replicas. For executing $x.decrement(n)$, the operation must acquire and consume n rights to decrement x in the replica it is submitted. If not enough rights exist in the replica, the system will try to obtain additional rights from other replicas. If this is not possible, the operation will fail. Executing $x.increment(n)$ creates n rights to decrement n, initially assigned to the replica in which the operation that executes the increment is submitted.

A similar approach is used for invariants of the form $x \leq k$, with increments consuming rights and decrements creating new rights. For invariants of the form $x+y+\ldots+z \geq k$, a single escrow reservation is used, with decrements to any of the involved variables consuming rights and increments creating rights. If a variable x is involved in more than one invariant, several escrow reservations will be affected by a single increment/decrement operation on x.

The variant called *escrow reservation for conditions* checks a count of elements against some condition; for instance, the number of participants in a tournament in the invariant $nrPlayers(T) < k$. In this case, if the same user is enrolled twice concurrently, two rights are consumed, although the number of participants increases by only one. This is conservative, but "leaks" rights. However, if the same user is disenrolled twice concurrently, then the number of users increases by only one; creating two rights might later let the invariant be violated.

Our escrow reservation for conditions addresses this problem using the following approach (considering invariant $c \geq k$). A decrement operation requires rights, just as a normal escrow reservation. However, an increment operation does not create rights immediately, but instead tags the reservation to be reevaluated. One of the replicas, marked as the primary for the reservation, is entrusted with recreating rights. To do so, it evaluates the distance between the current state and the threshold, taking into account the aggregate number of outstanding rights. More precisely, given the current value for $c = c_1$ and the number k_1 of outstanding rights (i.e., rights assigned to a replica and still not used, as known by the primary replica), $c_1 - k - k_1$ rights are created and assigned initially to the primary replica. This can be done either when the reservation is marked for reevaluation, or when new rights are needed.

Partition lock reservation: For some invariants, it is desirable to have the ability to reserve part of a partitionable resource. For example, consider the invariant that forbids two tournaments to overlap in time. Two operations that schedule different tournaments will break the invariant if the time periods overlap. Using a multi-level lock, it would be necessary to obtain an *exclusive allow* for executing any operation to schedule a new tournament.

However, no invariant violation arises if the time periods of concurrent operations do not overlap. To address this case, we provide a partition lock that allows a replica to obtain an *exclusive lock* on an interval of real values.[2] Replicas can obtain locks on multiple intervals, given that no two intervals reserved by different replicas overlap.

In our example, time would be mapped to a real number. To execute the operation that schedules a tournament, a replica would have to obtain a lock on an interval that includes the time from the start to the end of the tournament.

Using Reservations Our static analysis outputs *I-offender sets* and the corresponding invariant violated. A programmer, electing to use the conflict avoidance approach, must select the type of reservation to be used to avoid invariant violations. Figure 2 presents a default mapping between types of invariants and the corresponding reservations. Conservatively, it is always possible to resort to multi-level locks to enforce any invariant, at the expense of admissible concurrency, as discussed earlier. In the context of WP4, it has been shown how to prove that a system that uses a reservation system consisting only of multi-level locks preserves a given invariant [14].

After deciding which reservations will be used, each operation is extended to acquire the appropriate rights before executing its code, and to release appropriate

[2] Partition locks are a simplified version of partitionable objects [28] and slot reservations [21].

Invariant type	Formula (example)	Reservation
Numeric	$x < K$	Escrow(x)
Referential	$p(x) \Rightarrow q(x)$	Multi-level lock
Disjunction	$p_1 \vee \ldots \vee p_n$	Multi-level mask
Overlapping	$t(s_1, e_1) \wedge t(s_2, e_2) \Rightarrow s_1 \geq e_2 \vee e_1 \leq s_2$	Partition lock
Default	—	Multi-level lock

Table 2: Default mapping from invariants to reservations.

rights afterwards. For escrow locks, an operation that consumes rights will acquire rights before its execution (and these rights will not be released when the operation ends). Conversely, an operation that creates rights will create these rights after its execution. For multi-level masks, the programmer must provide the code that verifies the values of the predicate associated with each element of the disjunction.

4.2.2.2 Indigo System Our Indigo prototype of was extended to support the new reservations. The details of the implementation are described in Balegas et. al. [4]. The evaluation in a geo-replicated environment shows that the proposed approach can enforce application invariants while most operations complete in the local data center, thus providing a much lower latency than solutions requiring coordination among replicas.

4.2.3 Explicit Consistency with Invariant-repair

The results of our evaluation of Indigo show that most operation can execute locally. However, in some cases, the execution of some operations requires obtaining reservations from remote site. This leads to high latency, with operation execution taking even longer than in strong consistency settings, and lower availability as fault may make it impossible to obtain the necessary reservations.

To address this problem, we have been exploring two alternative techniques for enforcing invariants [5]. We now present such techniques in the context of an example application.

4.2.3.1 Example Application We will use as an example application, a tournament micro-service that can be used to support most common competition online games. In this application, players participate in tournaments and compete against each other in matches.

A tournament has three phases: an enrollment phase where players can enroll in the tournament, an active phase where there can be no modifications to the participants of the tournament and a finished phase, when the tournament is concluded and a winner is elected, based on the number of points achieved in each match. A tournament cannot be removed after it starts and has a minimum and a maximum number of participants. A tournament has a leader that can start or remove the tournament, the leadership role can be shared with other players. A player can deposit and spend credit anytime to buy items that are used in the game to get advantage over the adversary. Items have limited availability.

Example 1: A Matter of Ordering While a tournament does not start, players can enroll and disenroll, but the tournament can only start after a minimum number of players have enrolled in the tournament. When a partial ordering of execution is allowed, this constitutes a problem for invariant preservation: a leader of the tournament can start the tournament because he observer, in the local replica, that there is a minimum number of players enrolled, however, concurrently, at a remote replica, a player might disenroll from the tournament, dropping the number of players below minimum. Under serial execution this does not occur because one of the operations will fail, i.e., either the player cannot disenroll from the tournament, because it starts before, or the tournament cannot start because it does not have enough players. Despite the fact that serialization ensures the applications invariants, programmers need to check that the preconditions of the operations are met before modifying the state of the database.

Under partial ordering execution, the operations must also check the pre-conditions of the operations before taking any action locally, but that does not preclude a concurrent operation from interfering with this one. It might occur that a concurrent remote operation also satisfies its local dependencies but is conflicting with the current operation, and, when both operations are delivered in the same replica, an invariant violation occurs.

Different strategies to repair the invariant violation are possible: we can apply a repair function that makes none of the operations take effect; or the player is not disenrolled from the tournament and the tournament can start, or the player is disenrolled from the tournament and the tournament is canceled. The first solution does not provide a good user experience, because both users will see their actions retracted. The other two repair functions provide a semantic equivalent to the serializable execution, i.e. operations appear to have executed one after the other. However, there is an important caveat with this conflict resolution: more operations might depend on the operation being repaired, for instance, a player might have participated a match after the tournament had started and if we chose to cancel the tournament, in theory, that game could not have occurred usar este caso no explicit concurrency. In this case it is easy to stop invariant violation from contaminating other operations. We can chose to remove the player from the tournament, in which case no other operation is affected by this convergence policy because no other operation in the workload depends on the player not being enrolled in the tournament to be able to execute 1.

In general, it might be necessary to analyze conflict res- olution strategies in order to prevent the generation of new conflicts. We intend to study static analysis to evaluate the quality of repair strategies.

Example 2: When Ordering is not Enough In some situations, invariant violations are not easily repaired. Consider that two players concurrently bought the last unit of an item in the application. For this conflict we cannot apply a repair function that produces a state equivalent to one operation executing after the other, because one of the requests would have different effects, i.e. the operation would fail because there are no available resources left. This situation occurs when operations are not commutative, which means that we cannot arbitrate an ordering for their execution without producing different effects. This is different from the previous example because, in the first case, despite arbitrating the execution ordering of the pair of operations, the effects of both operations are preserved.

In fact, a serial execution is what makes most sense in the real life, as it would be impossible to duplicate resources. We could think of a service that allows items to be sold in parallel and therefore overselling, but we cannot take more items then physically available.

If this invariant is important for the application, we have no option then to use a strong coordination mechanism to ensure that no user buys more resources then available. However, some invariants, or lets say, application properties, are desirable properties and not essential for correctness, in which case more solutions are possible. To not be unfair with any player, the applications could allow the item to be sold twice which is equivalent to the semantics of eventual consistency. Or, remove the item from one of the player's inventory and give back some credit. In this case, she might have used the item already and that would create more conflicts. The developer can still make this choice, as long as she is able to repair any operation that used the resource. The last alternative is to create new items to compensate for the advantage that were given to both players.

Our conclusion is that some operations naturally require a coordinated execution, but one can make an alternative version of the same algorithm that does not require serialization and apply a compensation when things go wrong. This is how online stores deal with exhausted stocks, or ATMs handle withdrawals that cannot read the actual balance of an account, do in practice.

4.2.3.2 Invariant-aware Convergence Rules Consider that we repair the invariant violation of example 1 by keeping the player in the tournament. we pursue a repair strategy that does not impair the availability of the system, therefore we avoid strategies that assume a central authority or require coordination to ensure that the invariant is repaired.

The algorithm we propose is based on the convergence rules used in CRDTs[25]. CRDTs can ensure add-/remove- wins policies when concurrent add and remove operations execute over the same data-type. This means that we can select the outcome of a concurrent add/remove operation of the same element to a set. In the example, the begin operation checks that the set of participants in the tournament has the minimum number of players and then changes the value of some flag to true, meaning that the tournament has started. The concurrent disenroll operation removes one element from the participants set, making its size smaller then the minimum and when both operations are propagated to the same replica we end in a state with a set of participants that is smaller than the required for the value of the flag being true.

The solution for this problem is quite easy. Considering that the set of participants uses a add-wins strategy for handling conflicting adds and removes. This allows to ensure that that the size of the tournament does not decrease with any concurrent remove, because we can cancel the effect of the remove with an add. In order to do that, when starting the tournament, we just add again, to the set of participants, all the players that belong to the set of player in the moment the tournament starts. This enforces that any concurrent remove will take no effect, because the merge strategy of the set preserves the concurrent adds, therefore the size of the set does not decrease. Adding all the elements to the set again can be done in an efficient way, to avoid processing overheads when the tournament is large.

The benefit of this strategy is that it does not require any additional mechanism to detect conflicts, as the execution of the operations automatically enforces the pre-conditions for the operation hold when its delivered to any replica. This strategy additionally requires identifying the pre-conditions of operations, instrument the code with the extra updates and check for compatibility between operations, i.e., that the different convergence rules are compatible with the invariants. We recognize that it might not be possible to handle all conflicts with this strategy, but it is promising.

4.2.3.3 Compensating Conflicts In the second example, we describe the case where the invariant violation cannot be repaired, thus the system must handle it as part of an exception of the workload. To handle those situations we want to apply some action that compensates for the occurrence. In the previous example, the solutions consist in doing nothing, remove the item from one player's item list, or create new resources. However, two things have to be taken into consideration when writing compensations: does the compensation conflict with any other invariant? what happens if two different replicas compensate the same action? To answer the first question we can consider a compensating action as part of the workload and use the same tools to detect the conflicts of the application. Applying compensating actions is trivial when the operations are idempotent, because they can execute at multiple machines without producing further outputs. The problem with applying the compensating actions is that if when the operation is non-idempotent, in which case, multiple executions of the same operation produce cumulative effects. For instance, if the compensation was to create new resources, it could create more resources than desired.

The simplest way to implement the compensation mechanism is to use a central authority that would guarantee that the compensating action only occurred once, or use a consensus algorithm. However, this requires coordination which is what we are trying to avoid at all costs. The alternatively is to make compensations idempotent. To enable that, replicas need to maintain the log of the operation they applied, the information of what compensations they applied to solve each conflict and the compensating operations must be deterministic and independent of the current state of the database. If every replica keeps this information, they can independently identify what remote replicas have applied the compensation and cancel the effect of multiple repairs. The downside of the approach is that replicas cannot compress the log until all replicas have received the conflicting updates, but we can assume that partitions, when they occur, do not last forever. The damage of compressing the log before a replica acknowledges a conflict is measurable and the developer can decide to move forward after some time, to avoid the log size to increase. The result would be that the unreachable nodes could have compensated for the same conflict and the effects will accumulate. Another property of compensating transactions is that they may not have to be executed immediately, i.e., the system can delegate applying the fix to the future, which can be convenient in some cases.

4.2.3.4 Future Work

We are currently focusing on implementing these approaches. This work is being developed in context of both WP3 and WP4, with algorithm and protocol development being part of WP3 and language aspects being part of WP4.

Preliminary results with the development of our first approach E.1 show that this approach can address the problem of operations with high latency at the cost of increasing the latency in all other operations. Although this increase is small, we are currently studying how to avoid this effect.

4.3 Extensions to Works Previously Reported

During this period, as planned, we have also started working on security (as part of Task 3.3). We have proposed a first initial algorithm for managing access control information and enforcing access control. This proposal addresses only a limited setting and will be extended in the next period. This work will be reported in the following period.

A number of works that started being developed in the context of Task 3.1 have continued during this period, some of them leading to publications. We now briefly overview the most relevant work, some of them being developed jointly with other Work Packages.

4.3.0.5 Delta State-based CRDTs

Delta State-based CRDTs provide an efficient mechanism for synchronizing state-based CRDTs, by propagating only deltas among replicas. A Delta State-based CRDTs (δ-CRDT) is composed by: a state that is a join-semilattice that results from the join of multiple fine-grained states, i.e., deltas, generated by what we call δ-mutators; these are new versions of the datatype mutators that return the effect of these mutators on the state. In this way, deltas can be retained in a buffer to be shipped individually (or joined in groups) instead of shipping the entire object. The changes to the local state are then incorporated at other replicas by joining the shipped deltas with their own states.

In this period we continued developing this work [1] and a reference implementation in C++ that is publicly available [3].

4.3.0.6 Conflict-free Partially Replicated Data Types

A Conflict-free Partially Replicated Data Structure (CPRDTs) is a CRDT that can be partitioned in multiple *particles*. We define particles as the smallest meaningful elements of a CPRDT. By meaningful we refer to the smallest element that can be used for query and update operations. For instance, a particle in a grow-only set would be any element that can be added or looked up in the set.

In this period we have continued developing this work that has been recently accepted for publication [12].

4.3.0.7 Mechanisms for Efficient Transactions

[3] `https://github.com/CBaquero/delta-enabled-crdts`

Minimizing Conflicts in Transactions Over Partitioned Data A key characteristic of distributed transactional protocols that impacts the performance of transactional cloud data stores is the abort rate, which is affected by the degree of concurrency, and naturally, by the workload characteristics. However, the concurrency control mechanism may also play a fundamental role in reducing or increasing the likelihood of conflicts.

We define the vulnerability window as the time window defined between transaction's starting point and its serialization point. Two transactions whose vulnerability windows overlap may potentially cause one of the transactions to abort. In protocols that use clocks, as the algorithms being developed in the context of Antidote, the vulnerability window depends on how the protocol handles time.

In this work we propose a novel technique that aims at reducing the vulnerability window of transactions. Our technique uses an hybrid clock implementation. The idea is to use the physical part of the hybrid clock to set the starting time of the transaction; therefore, moving the starting point forward in time as much as possible. On the other hand, our technique proposes to use the logical part of the hybrid clock in order to serialize transactions at the earliest possible point in time. The combination of these two techniques has the potential of reducing the vulnerability window; and in consequence, the abort rate.

An initial report on this work as been presented in Bravo et. al. [11]. We expect to be able to apply the ideas presented in the protocols used in Antidote.

Concise Server-Wide Causality Management In the past we developed a number of technique for tracking causality in cloud stores, notably Dotted Version Vectors. Our previous approach, although improving on the state of the art, still incur in a non-negligible metadata overhead per key, which also keeps growing with time, proportional with the node churn rate. Another challenge is deleting keys while respecting causality: while the values can be deleted, per-key metadata cannot be permanently removed without coordination.

In this work [13] we propose a new causality management framework for eventually consistent data stores, that leverages node logical clocks (Bitmapped Version Vectors) and a new key logical clock (Dotted Causal Container) to provide advantages on multiple fronts: 1) a new efficient and lightweight anti-entropy mechanism; 2) greatly reduced per-key causality metadata size; 3) accurate key deletes without permanent metadata.

4.3.0.8 Generic Exactly-once Quantity Transfer We have proposed a generic algorithm for exactly-once transfer of a "quantity" from one node to another on an unreliable network and without any form of global synchronization [26]. Our approach copes with message duplication, loss, or reordering. This allows preserving a global property (the sum of quantities remains unchanged) without requiring global linearizability and only through using pairwise interactions between nodes, therefore allowing partitions in the system.

This technique can be used in CRDT used for maintaining invariants, such as the Bounded Counter, which need to transfer rights among replicas.

4.3.0.9 Efficient Support of Large CRDTs in Riak The deployment of CRDTs in Riak 2.0, and their use in multiple production scenario has led to unanticipated usage, such as clients storing huge number of elements inside sets. This has led to performance problems, as any operation on a set would need to access a complete replica of the set. Additionally, when a set is modified and needs to be propagated over the network to update remote replicas, the full replica needs to be transmitted.

Basho has been addressing this problems by redesigning the support for CRDTs inside Riak. In this work, described by Russell [4], the ideas previously proposed in Delta State-based CRDTs are being explored.

[4] `https://gist.github.com/russelldb/1dc3fde55f856833b18e`

5 Publications and presentations

The work performed in the context of WP3 and in collaboration with other work packages has led to several papers. The following papers have been published during this period:

- [1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation. In Proceedings NETYS 2015, Lecture Notes on Comptuer Science. Springer, 2015.
- [4] Valter Balegas, Sérgio Duarte, Carlos Ferreira, Rodrigo Rdorigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting the Consistency back into Eventual Consistency. In Proceedings of the sixth conference on Computer systems, EuroSys '15. ACM, 2015.
- [6] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In Proc. of the Symposium on Reliable Distributed Systems (SRDS'15), Set 2015.
- [3] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Towards Fast Invariant Preservation in Geo-replicated Systems. SIGOPS Oper. Syst. Rev., 49(1):121–125, January 2015.
- [11] Manuel Bravo, Paolo Romano, Luís Rodrigues, and Peter Van Roy. Reducing the Vulnerability Window in Distributed Transaction Protocols. In Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC '15. ACM, 2015.
- [13] Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero Moreno, and Vitor Fonte. Concise Server-Wide Causality Management for Eventually Consistent Data Stores. In Proceedings of the 15th International Conference on Distributed Applications and Interoperable Systems (DAIS'15), volume 9038 of Lecture Notes on Comptuer Science, Berlin, Germany, 2015. Springer, Springer.
- [26] Ali Shoker, Paulo Sérgio Almeida, and Carlos Baquero. Exactly-Once Quantity Transfer. In Proc. of the Workshop on Planetary Scale Distributed Systems 2015 (part of SRDSW), Sep 2015.

The following papers have been accepted and will be published during the next period:

- [12] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free Partially Replicated Data Types. In Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015). IEEE, Nov 2015.

The following paper are under submission or being prepared for submission.

- [9] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Submitted to ACM Queue, 2015.
- [7] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. Composition of State-based CRDTs. Technical report, U. Minho, 2015.

Besides presentation of accepted papers, the following works have been presented without formal proceedings.

- Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira. The Quest For Coordination Free Cloud Storage Systems. Poster accepted for presentation at SOSP'15.
- [5] Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, and Carla Ferreira. Designing concurrency-aware geo-replicated storage systems. Presented at the Workshop on Planetary Scale Distributed Systems 2015, 2015.
- Carla Ferreira. Putting the Consistency back into Eventual Consistency. Presented at FRIDA 2015, 2015.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation. In *Proceedings NETYS 2015*, Lecture Notes on Comptuer Science. Springer, 2015.

[2] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[3] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Towards Fast Invariant Preservation in Geo-replicated Systems. *SIGOPS Oper. Syst. Rev.*, 49(1):121–125, January 2015.

[4] Valter Balegas, Sérgio Duarte, Carlos Ferreira, Rodrigo Rdorigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting the Consistency back into Eventual Consistency. In *Proceedings of the sixth conference on Computer systems*, EuroSys '15. ACM, 2015.

[5] Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, and Carla Ferreira. Designing concurrency-aware geo-replicated storage systems. Presented at the Workshop on Planetary Scale Distributed Systems 2015, 2015.

[6] 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.

[7] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. Composition of State-based CRDTs. Technical report, U. Minho, 2015.

[8] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, volume 8460 of *Lecture Notes in Computer Science*, pages 126–140. Springer Berlin Heidelberg, 2014.

[9] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Submitted to ACM Queue, 2015.

[10] 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.

[11] Manuel Bravo, Paolo Romano, Luís Rodrigues, and Peter Van Roy. Reducing the Vulnerability Window in Distributed Transaction Protocols. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15. ACM, 2015.

[12] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, Nov 2015.

[13] Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero Moreno, and Vitor Fonte. Concise Server-Wide Causality Management for Eventually Consistent Data Stores. In *Proceedings of the 15th International Conference on Distributed Applications and Interoperable Systems (DAIS'15)*, volume 9038 of *Lecture Notes on Comptuer Science*, Berlin, Germany, 2015. Springer, Springer.

[14] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. Submitted for publication, 2015.

[15] Irum Kazmi and Syed Fahim Yousaf Bukhari. Peersim: An efficient & scalable testbed for heterogeneous cluster-based p2p network protocols. In *Proceedings of the 2011 UKSim 13th International Conference on Modelling and Simulation*, UKSIM '11, pages 420–425, Washington, DC, USA, 2011. IEEE Computer Society.

[16] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[17] Narayanan Krishnakumar and Ravi Jain. Escrow techniques for mobile sales and inventory applications. *Wirel. Netw.*, 3(3):235–246, August 1997.

[18] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[19] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 503–517, Berkeley, CA, USA, 2014. USENIX Association.

[20] Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[21] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.

[22] Calton Pu, Wenwey Hseush, Gail E Kaiser, Kun-Lung Wu, and Philip S Yu. Distributed divergence control for epsilon serializability. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 449–456. IEEE, 1993.

[23] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, December 1995.

[24] Nuno Santos, Luís Veiga, and Paulo Ferreira. Vector-field consistency for adhoc gaming. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 80–100, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[25] 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.

[26] Ali Shoker, Paulo Sérgio Almeida, and Carlos Baquero. Exactly-Once Quantity Transfer. In *Proc. of the Workshop on Planetary Scale Distributed Systems 2015 (part of SRDSW)*, Sep 2015.

[27] Liuba Shrira, Hong Tian, and Doug Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[28] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–, Washington, DC, USA, 1995. IEEE Computer Society.

[29] Kun-Lung Wu, Philip S. Yu, and Calton Pu. Divergence control algorithms for epsilon serializability. *IEEE Trans. on Knowl. and Data Eng.*, 9(2):262–274, March 1997.

[30] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

A Estimating Probabilistic Divergence Metrics

A.1 System Model

Consider a replicated system of n replicas distributed over m groups (say DCs). Replicas r_{ik} and r_{il} within group g_i communicate with each other every period T^t_{kl}. We use the notation r_k to refer to a replica regardless of any grouping. Groups can also communicate with each others through *group leaders*. A leader acts as a synchronization portal between its own group replicas and other groups. Two leaders l_i and l_j (or simply groups g_i and g_j) communicate with each other, less frequently, on every period T^t_{ij}. Leaders can be chosen using any existing voting method, and it is orthogonal to this work.

A replica can host one or more CRDTs. (We use the term "replica" in this paper to represent the datatype copy itself, otherwise else mentioned.) A replica is thus allowed to update its local state with a loose coordination with other replicas, which leads to temporary divergence. Conflicts between diverging replicas are eventually resolved as soon as replicas coordinate. We assume that this conflict resolution is a black box, and we are only interested in the rate of updates on the replicated datatype. The rate of updates of a replica r_{ik} (resp., group g_i) is R^T_{ik} (resp., R^T_i) during a period T.

A.2 System Topology

To improve scalability, geo-replicated systems are often composed of few groups, e.g., Data centers (DCs), comprising plenty of replicas. In this topology, intra-group coordination is more frequent than inter-group on. Since it is not practical for every single replica to coordinate with all other system replicas, we assume that each group has a leader than can coordinate, on behalf of his group, with other group leaders. In this design, a leader is in charge of pushing/pulling the updates across groups. Group members are informed about the progress of remote replicas (in other groups) through their leader. As shown in Figure..., leaders can be seen as an additional virtual group, but often with looser coordination.

A.3 Divergence Assessment

As mentioned above, replicas are allowed to diverge between synchronization periods. That said, it is crucial for many applications to assess the progress (mainly updates) of the CRDT state until coordination occurs in the next period, e.g., in order to avoid violations or make convenient actions. To do this, a replica r_i can retain a vector V_i of the update rates all other replicas. On each coordination period T^t_i with another replica r_j, the joint rate $R^{T^t_{ij}}_{ij}$ is updated. Many tradeoffs exist for maintaining the rate's vector V; we discuss three of them in what follows:

Vector Size. Assume that the size of V is an order of n (the number of replicas). As the number of replicas gets large, maintaining V becomes costly as the transmission and space overheads will grow. In addition, each replica will be coordinating wit all other replicas which is not practical. On the opposite side, if V only retains the rates of groups (an order of m); then the aggregated rate of changes by remote groups would be high; put together with the low coordination frequency across groups, this will significantly reduce the accuracy of estimating the evolution of remote replicas, and consequently lead to violations. The alternative tradeoff is to retain the rate of neighboring replicas, i.e., those within the same group, in addition to those of the leaders of remote groups. This will be a more scalable solution that has an acceptable level of freshness needed for estimation. (TODO: it is good to experiment this tradeoff).

Driven by the above discussion, we require the vector of rates to concatenate two vectors, i.e., $V = V_r|V_g$, where V_r retains the rates on intra-group replicas and V_g retains those of groups. In the following, we address V as a single vector as long as discriminating V_r and V_g is indifferent.

Coordination Rate. A second tradeoff that can impact the divergence of the system is the coordination rate. A frequently coordinated system is costly and hard to maintain in the presence of partitions. On the contrary, very loosely coordinated system can often lead to breaking the prospective invariants of the system. Distributed the replicas over groups with different coordination periods partially resolves this problem as it imposes a more frequent coordination for nearby replicas than remote ones. A question arises here is how to tune the coordination periods and whether they shall be static or dynamic? In the following section, we try to give a solution based on statically chosen coordination periods and also how to efficiently choose dynamic coordination periods among groups and also individual replicas.

Checking for Violations. One more tradeoff is how greedy the system would be in checking for violations. One option is to check for violations before each update is executed. An update is successful if it does not lead to violations, otherwise an action must be made (e.g., stop, require coordination now, use escrow, etc.). The other option is using this checking only when it is likely that violations will occur (with some probability); this allows to skip violation checking as long as the state of the CRDT is far from invariant breaking. An example is when a counter CRDT (e.g., an Ad) is quite large to be consumed upon many decrements before coordination. The third option is to dynamically change the coordination periods, based on the rates in V, in such a way to avoid violations. In this way, replicas can blindly execute their local updates without checking for other replicas progress. A replica is only required to require coordination if its own local rate of updates in the current coordination period is much higher than that of previous periods. The solution we provide in this work is based on this final option. (TODO: We plan to consider the other options in the future.).

Since different CRDT datatypes diverge in different ways, we address each datatype aside. We give a more emphasis on the “counter” datatype that has many use cases in practice. Furthermore, it is easy to get inspired by this approach on counters to assess the divergence of “maps”, set sizes, etc.

A.4 Counters

A counter has two main update operations: an increment and decrement. For simplicity, we only consider a counter $c > 0$ and a single decrementing operation: dec (e.g., Ads use-case). It is straightforward to apply the same techniques (described next) on increments too. A replica c_i of a counter c can be decremented d times if this does not break the invariants of the counter, i.e., $c \nless 0$. To ensure this, at any time, a replica must test whether d decrements would lead to violation or not. If the replica has an exact value c of the counter, it can avoid violations by simply asserting $c - d < 0$. However, since an exact value c is not possible to compute in practice, a replica can compute an estimate $\bar{c}$ of it instead. Since a local counter value c_i is always accurate, thus this replica can estimate

$$\bar{c} = c_i - \sum_{j \neq i} \bar{d}_j \tag{12}$$

where $\bar{d}_j$ is the estimated number of decrements on replica j. The challenge is now how to compute this estimate. Let t^p_{ij} and t^n_{ij} be the previous and next coordination time, respectively, between replicas i and j. Let c^t_i be the local counter value of any replica i at time $t^p_{ij} < t < t^n_{ij}$. Replica i can estimate the number of decrements $\bar{d}_j$ on another replica j in three ways: direct, probabilistic, or optimistic.

A.4.1 Direct Estimation

The direct estimation of $\bar{d}_j$ at replica i is a simple but inaccurate method that depends on the previous rate value of $V_i[j]$ and the duration since the previous coordination:

$$\bar{d}_j = (t - t_p)V_i[j] \tag{13}$$

The above method is straightforward and cheap; however, it is only applicable if the rates are merely constant over time. Since this is not very realistic in many applications, it is good to include a variation error for this estimate. We follow the probabilistic approach to calculate this error.

A.4.2 Probabilistic Estimation

In this approach, we assume that the number of decrements, i.e., the rate, follows a probability distribution D. Then we try to estimate an interval in which the estimated number of decrements lie with a high probability, and we need to find the probability that a number of potential local decrements can lead to violation.

A.4.2.1 Confidence Intervals A frequently used method is to find the confidence interval considering high confidence percentages, e.g., 95% and 99%, in which the "mean" value μ_i would lie. Such interval often satisfies is or approximated by the Z- or t-distributions (regardless of D). The estimated interval of decrements is then as follows:

$$\bar{d}_i = [\check{d}_i, \hat{d}_i] = \left[\mu_i \pm \tau \frac{\sigma_i}{\sqrt{k_i}}\right] \tag{14}$$

Table 3: The values of τ as confidence varies.

Confidence percentage	95%	99%	99.9%
$\tau \backsim$ Z-dist	1.96	2.58	3.39
$\tau \backsim$ t-dist			

where μ_i and σ_i are the mean and the variance of the probability distribution D_i on replica i, and τ (see Table 3) is the Z- or t-score corresponding to the confidence percentage required (more details to come next). In addition, k_i represents the number of coordinations replica i made with the local replica. Eq. 14 is used to compute the confidence interval in one time slot according to the rates. For instance, if the rates are calculates per seconds, then this estimate would be for only one second ahead. But how could we calculate this for longer periods?

In fact, we can benefit from the fact that the rate of decrements of counters is independent across time slots (e.g., the number of decrements of a slot does not impact the number of decrements of the next one). Given this, calculating the interval on $t - t_p$ time slots can be seen as computing the interval of the sum of $t - t_p$ independent distributions. Since the interesting statistical rules say that the mean μ and variance σ^2 of the sum of independent distributions add, then we get the confidence interval of the number of decrements during $(t - t_p)$ as follows:

$$\bar{d}_i^t = [\check{d}_i^t, \hat{d}_i^t] = \left[\mu_i(t - t_p) \pm \tau\sqrt{(t - t_p)\frac{\sigma_i}{\sqrt{k_i}}}\right] \tag{15}$$

It is clear that Eq. 15 depends on $t - t_p$. Since t_p is fixed, then as t gets larger more decrements are expected and also more error is expected (the second part of the equation). This is reasonable since as the coordination time increases (and equivalently the coordination period), the estimation error would escalate due to potential variation in the decrementing rate on remote replicas. Eq. 15 depends also on k_i which is the number of coordinations occurred so far. Notice that as k_i increases the estimation error decreases. This is referred to the increasing confidence gained by using μ_i (which is often the mean of $V[i]$ of the k_i previous coordination periods).

Back to Eq. 12, we now introduce the confidence interval of the counter at any time t:

$$\bar{c}_i^t = [\check{c}_i^t, \hat{c}_i^t] = \left[c_i^t - \sum_{j \neq i} \hat{d}_j^t,\ c_i^t - \sum_{j \neq i} \check{d}_j^t\right] \tag{16}$$

A.4.2.2 Detecting Violations In this section, we show how to compute the probability of a violation to occur upon d decrements. Upon d decrements executed by the local replica i, a violation occurs if the counter value becomes negative, i.e., $c - d < 0$. But, $c = c_i - \bar{d}$, where $\bar{d}$ is the estimated decrements on all groups. Therefore, in general, the probability of a violation to happen upon d decrements is:

$$P(c - d < 0) = P(c_i - \bar{d} - d < 0) = P(\bar{d} > c_i - d) = 1 - P(\bar{d} \leq c_i - d)$$

Given that $\bar{d}$ follows the probability distribution D, it is easy to find this commutative probability.

Another option to check for violation is possible in case we have a pre-defined probability as revealed in Table 3. This is useful if the estimated counter value $\bar{c}_i^t$ is already computed. For instance of it is accepted to check violation with probability 97.5%, then we can use $\tau = 1.96$ and assert that

$$\hat{c}_i^t - d \nless 0. \tag{17}$$

A.4.2.3 Non-Greedy Checking The above violation checking and interval estimations are very demanding to computation resources as computing the probabilities must be done on each attempt to decrement locally. A more practical way is to follow a non-greedy way in which we can check the possibility of violations only once, just after entering a new coordination period. This can be simply done by fixing t and using T_n instead. This can be viewed as an escrow over the current coordination period $T_n - t_p$.

This approach can be used with confidence as long as the probability of violating the invariants is close to zero, i.e., the counter is far from zero. However, as the probability of violations start to increase, it can be useful to degrade the coordination frequency (e.g., an exponential back-off) in favor of non-greedy checking. This would mean that as the counter value approaches to zero, the coordination periods will get smaller, and consequently replicas must coordinate faster. This can sometimes be undesirable. A more smart solution is to compute the coordination periods using a probabilistic back-off scheme.

A.4.3 Example: Poisson Distribution

More details to come, but briefly. If D is the Poisson distribution, which is often used for similar situations, then $\mu_i = \sigma_i = V[i]/k_i$.

An interesting feature in Poisson is also that the sum of independent Poisson distribution follows Poisson of the sum of their means, i.e., $P(\lambda_1, \lambda_2, \ldots, \lambda_n) = P(\lambda_1 + \lambda_2 + \cdots + \lambda_n)$.

Thus $\mu_{j\neq i} = \sum_{j\neq i} \mu_j$ and $\lambda_{j\neq i} = \sum_{j\neq i} \lambda_j$. Then the sum of estimated decrements on all other replicas becomes:

$$\bar{d}_{j\neq i}^t = [\check{d}_{j\neq i}^t, \hat{d}_{j\neq i}^t] = \left[\mu_{j\neq i}(t - t_p) \pm \tau \sqrt{(t - t_p)} \frac{\sigma_{j\neq i}}{\sqrt{k_{j\neq i}}} \right] \tag{18}$$

where $k_{j\neq i}$ can be computed as the median or mean of all ks.

And Eq. 16 becomes:

$$\bar{c}_i^t = [\check{c}_i^t, \hat{c}_i^t] = \left[c_i^t - \hat{d}_{j\neq i}^t,\ c_i^t - \check{d}_{j\neq i}^t \right] \tag{19}$$

B Published papers

B.1 Paulo Sérgio Almeida, Ali Shoker, Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation. In Proc. NETYS 2015, 2015. Springer.

Efficient State-based CRDTs by Delta-Mutation

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero

HASLab/INESC TEC and Universidade do Minho, Braga, Portugal
{psa,shokerali,cbm}@di.uminho.pt *

Abstract. CRDTs are distributed data types that make eventual consistency of a distributed object possible and non ad-hoc. Specifically, state-based CRDTs ensure convergence through disseminating the entire state, that may be large, and merging it to other replicas; whereas operation-based CRDTs disseminate operations (i.e., small states) assuming an exactly-once reliable dissemination layer. We introduce *Delta State Conflict-Free Replicated Datatypes* (δ-CRDT) that can achieve the best of both worlds: small messages with an incremental nature, disseminated over unreliable communication channels. This is achieved by defining *δ-mutators* to return a *delta-state*, typically with a much smaller size than the full state, that is joined to both: local and remote states. We introduce the δ-CRDT framework, and we explain it through establishing a correspondence to current state-based CRDTs. In addition, we present an anti-entropy algorithm that ensures causal consistency, and two δ-CRDT specifications of well-known replicated datatypes.

Keywords: Replicated data types; state-based CRDT; delta mutation.

1 Introduction

Eventual consistency (EC) is a relaxed consistency model that is often adopted by large-scale distributed systems [11,24,13] where availability must be maintained, despite outages and partitioning, whereas delayed consistency is acceptable. A typical approach in EC systems is to allow replicas of a distributed object to temporarily diverge, provided that they can eventually be reconciled into a common state. To avoid application-specific reconciliation methods, costly and error-prone, *Conflict-Free Replicated Data Types* (CRDTs) [22,23] were introduced, allowing the design of self-contained distributed data types that are always available and eventually converge when all operations are reflected at all replicas. Though CRDTs are being deployed in practice [11], more work is still required to improve their design and performance.

CRDTs support two complementary designs: *operation-based* (or op-based) and *state-based*. In op-based designs [17,23], the execution of an operation is

* This work is co-financed by the North Portugal Regional Operational Programme (ON.2, O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), within project NORTE-07-0124-FEDER-000058; and by EU FP7 SyncFree project (609551).

done in two phases: *prepare* and *effect*. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the operation is applied remotely using *effect*. On the other hand, in a state-based design [4,23] an operation is only executed on the local replica state. A replica periodically propagates its local changes to other replicas through shipping its entire state. A received state is incorporated with the local state via a *merge* function (designed as a least upper bound over a join-semilattice [4,23]) that deterministically reconciles both states.

Op-based CRDTs have more advantages as they can allow for simpler implementations, concise replica state, and smaller messages; however, they are subject to some limitations: First, they assume a message dissemination layer that guarantees reliable exactly-once causal broadcast (required to ensure idempotence); these guarantees are hard to maintain since large logs must be retained to prevent duplication even if TCP is used [15]. Second, membership management is a hard task in op-based systems especially once the number of nodes gets larger or due to churn problems, since all nodes must be coordinated by the middleware. Third, the op-based approach requires operations to be executed individually (even when batched) on all nodes.

The alternative is to use state-based systems which are deprived from these limitations. However, a major drawback in current state-based CRDTs is the communication overhead of shipping the entire state, which can get very large in size. For instance, the state size of a *counter* CRDT (a vector of integer counters, one per replica) increases with the number of replicas; whereas in a *grow-only Set*, the state size depends on the set size, that grows as more operations are invoked. This communication overhead limits the use of state-based CRDTs to data-types with small state size (e.g., counters are reasonable while sets are not). Recently, there has been a demand for CRDTs with large state sizes (e.g., in RIAK DT Maps [6] that can compose multiple CRDTs).

In this paper, we rethink the way state-based CRDTs should be designed, having in mind the problematic shipping of the entire state. Our aim is to ship a *representation of the effect* of recent update operations on the state, rather than the whole state, while preserving the idempotent nature of *join*. This ensures convergence over unreliable communication (on the contrary to op-based). To achieve this, we introduce *Delta State-based CRDTs* (δ-CRDT): a state is a join-semilattice that results from the join of multiple fine-grained states, i.e., *deltas*, generated by what we call *δ-mutators* which are new versions of the datatype mutators that return the effect of these mutators on the state. Thus, deltas can be temporarily retained in a buffer to be shipped individually (or joined in groups) instead of shipping the entire object. The local changes are then incorporated at other replicas by joining the shipped deltas with their own states.

The use of "deltas" (i.e., incremental states) may look intuitive in state dissemination; however, this is not the case for state-based CRDTs. The reason is that once a node receives an entire state, merging it locally is simple since there is no need to care about causality, as both states are self-contained (including meta-data). The challenge in δ-CRDT is that individual deltas are now "state fragments" and must be causally merged to maintain the correct semantics. This raises the following questions: is merging deltas semantically equivalent to merging entire states in CRDTs? If not, what are the sufficient conditions to make this true in general? And under what constraints causal consistency is maintained? This paper answers these questions and presents corresponding solutions.

We address the challenge of designing a new δ-CRDT that conserves the correctness properties and semantics of an existing CRDT by establishing a relation between the novel δ-mutators with the original CRDT mutators. We then show how to ensure causal consistency using deltas through introducing the concept of *delta-interval* and the *causal delta-merging condition*. Based on these, we then present an anti-entropy algorithm for δ-CRDT, where sending and then joining delta-intervals into another replica state produces the same effect as if the entire state had been shipped and joined.

As the area of CRDTs is relatively new, we illustrate our approach by explaining a simple *counter* δ-CRDT specification; then we introduce a challenging non-trivial specification for a widely used datatype: Optimized Add-Wins Observed-Remove Sets [5]; and finally we present a novel design for an Optimized Multi-Value Register with meta-data reduction. In addition, we make a basic δ-CRDT C++ library available online [2] for various CRDTs: GSet, 2PSet, GCounter, PNCounter, AWORSet, RWORSet, MVRegister, LWWSet, etc. Our experience shows that a δ-CRDT version can be devised for most CRDTs, however, this requires some design effort that varies with the complexity of different CRDTs. This is referred to the ad-hoc way CRDTs are designed in general (which is also required for δ-CRDTs). To the best of our knowledge, no model has been introduced so far to make designing CRDTs generic rather than type-specific.

2 System Model

Consider a distributed system with nodes containing local memory, with no shared memory between them. Any node can send messages to any other node. The network is asynchronous; there is no global clock, no bound on the time a message takes to arrive, and no bounds are set on relative processing speeds. The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted). Some messages will, however, eventually get through: if a node sends infinitely many messages to another node, infinitely many of these will be delivered. In particular, this means that there can be arbitrarily long partitions, but these will eventually heal. Nodes have access to durable storage; they can crash but will eventually recover with the content of the durable storage just before crash the occurred. Durable state is written atomically at each state transition. Each node has access to its globally unique identifier in a set $\mathbb{I}$.

3 A Background of State-based CRDTs

Conflict-Free Replicated Data Types [22,23] (CRDTs) are distributed datatypes that allow different replicas of a distributed CRDT instance to diverge and ensures that, eventually, all replicas converge to the same state. State-based CRDTs achieve this through propagating updates of the local state by disseminating the entire state across replicas. The received states are then merged to remote states, leading to convergence (i.e., consistent states on all replicas).

A state-based CRDT consists of a triple (S, M, Q), where S is a join-semilattice [12], Q is a set of query functions (which return some result without modifying the state), and M is a set of mutators that perform updates; a mutator $m \in M$ takes a state $X \in S$ as input and returns a new state $X' = m(X)$. A join-semilattice is a set with a *partial order* $\sqsubseteq$ and a binary *join* operation $\sqcup$ that returns the *least upper bound* (LUB) of two elements in S; a *join* is designed to be commutative, associative, and idempotent. Mutators are defined in such a way to be *inflations*, i.e., for any mutator m and state X, the following holds:

$$X \sqsubseteq m(X)$$

In this way, for each replica there is a monotonic sequence of states, defined under the lattice partial order, where each subsequent state subsumes the previous state when joined elsewhere.

Both query and mutator operations are always available since they are performed using the local state without requiring inter-replica communication; however, as mutators are concurrently applied at distinct replicas, replica states will likely diverge. Eventual convergence is then obtained using an *anti-entropy* protocol that periodically ships the entire local state to other replicas. Each replica merges the received state with its local state using the *join* operation in S. Given the mathematical properties of *join*, if mutators stop being issued, all replicas eventually converge to the same state. i.e. the least upper-bound of all states involved. State-based CRDTs are interesting as they demand little guarantees from the dissemination layer, working under message loss, duplication, reordering, and temporary network partitioning, without impacting availability and eventual convergence.

Example. Fig. 1 represents a state-based increment-only counter. The CRDT state Σ is a map from replica identifiers to positive integers. Initially, σ_i^0 is an empty map (assuming that unmapped keys implicitly map to zero, and only non zero mappings are stored). A single mutator, i.e., inc, is defined that increments the value of the local replica i (returning the updated map). The query operation value returns the counter value

$$\begin{aligned}
\Sigma &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\sigma_i^0 &= \{\} \\
\mathsf{inc}_i(m) &= m\{i \mapsto m(i) + 1\} \\
\mathsf{value}_i(m) &= \sum_{i \in \mathbb{I}} m(i) \\
m \sqcup m' &= \{(i, \mathsf{max}(m(i), m'(i))) \mid i \in \mathbb{I}\}
\end{aligned}$$

Fig. 1: State-based Counter CRDT; replica i.

by adding the integers in the map entries. The join of two states is the point-wise maximum of the maps.

Weaknesses. The main weakness of state-based CRDTs is the cost of dissemination of updates, as the full state is sent. In this simple example of counters, even though increments only update the value corresponding to the local replica i, the whole map will always be sent in messages though the other map values remained intact (since no messages have been received and merged).

It would be interesting to only ship the recent modification incurred on the state. This is, however, not possible with the current model of state-based CRDTs as mutators always return a full state. Approaches which simply ship operations (e.g., an "increment n" message), like in operation-based CRDTs, require reliable communication (e.g., because increment is not idempotent). In contrast, our approach allows producing and encoding recent mutations in an incremental way, while keeping the advantages of the state-based approach, namely the idempotent, associative, and commutative properties of join.

4 Delta-state CRDTs

We introduce *Delta-State Conflict-Free Replicated Data Types*, or δ-CRDT for short, as a new kind of state-based CRDTs, in which *delta-mutators* are defined to return a *delta-state*: a value in the same join-semilattice which represents the updates induced by the mutator on the current state.

Definition 1 (Delta-mutator). *A delta-mutator m^δ is a function, corresponding to an update operation, which takes a state X in a join-semilattice S as parameter and returns a delta-mutation $m^\delta(X)$, also in S.*

Definition 2 (Delta-group). *A delta-group is inductively defined as either a delta-mutation or a join of several delta-groups.*

Definition 3 (δ-CRDT). *A δ-CRDT consists of a triple (S, M^δ, Q), where S is a join-semilattice, M^δ is a set of delta-mutators, and Q a set of query functions, where the state transition at each replica is given by either joining the current state $X \in S$ with a delta-mutation:*

$$X' = X \sqcup m^\delta(X),$$

or joining the current state with some received delta-group D:

$$X' = X \sqcup D.$$

In a δ-CRDT, the effect of applying a mutation, represented by a delta-mutation $\delta = m^\delta(X)$, is decoupled from the resulting state $X' = X \sqcup \delta$, which allows shipping this δ rather than the entire resulting state X'. All state transitions in a δ-CRDT, even upon applying mutations locally, are the result of some join with the current state. Unlike standard CRDT mutators, delta-mutators do not need to be inflations in order to inflate a state; this is however ensured by joining their output, i.e., deltas, into the current state.

In principle, a delta could be shipped immediately to remote replicas once applied locally. For efficiency reasons, multiple deltas returned by applying several delta-mutators can be joined locally into a delta-group and retained in a buffer. The delta-group can then be shipped to remote replicas to be joined with their local states. Received delta-groups can optionally be joined into their buffered delta-group, allowing transitive propagation of deltas. A full state can be seen as a special (extreme) case of a delta-group.

If the causal order of operations is not important and the intended aim is merely eventual convergence of states, then delta-groups can be shipped using an unreliable dissemination layer that may drop, reorder, or duplicate messages. Delta-groups can always be re-transmitted and re-joined, possibly out of order, or can simply be subsumed by a less frequent sending of the full state, e.g. for performance reasons or when doing state transfers to new members. Due to space limits, we only address causal consistency in this paper, while information about state convergence can be found in the associated technical report [1].

4.1 Delta-state decomposition of standard CRDTs

A δ-CRDT (S, M^δ, Q) is a *delta-state decomposition* of a state-based CRDT (S, M, Q), if for every mutator $m \in M$, we have a corresponding mutator $m^\delta \in M^\delta$ such that, for every state $X \in S$:

$$m(X) = X \sqcup m^\delta(X)$$

This equation states that applying a delta-mutator and joining into the current state should produce the same state transition as applying the corresponding mutator of the standard CRDT.

Given an existing state-based CRDT (which is always a trivial decomposition of itself, i.e., $m(X) = X \sqcup m(X)$, as mutators are inflations), it will be useful to find a non-trivial decomposition such that delta-states returned by delta-mutators in M^δ are smaller than the resulting state:

$$\mathsf{size}(m^\delta(X)) \ll \mathsf{size}(m(X))$$

4.2 Example: δ-CRDT Counter

Fig. 2 depicts a δ-CRDT specification of a counter datatype that is a delta-state decomposition of the state-based counter in Fig. 1. The state, join and value query operation remain as before. Only the mutator inc^δ is newly defined, which increments the map entry corresponding to the local replica

$$\begin{aligned}
\Sigma &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\sigma_i^0 &= \{\} \\
\mathsf{inc}_i^\delta(m) &= \{i \mapsto m(i) + 1\} \\
\mathsf{value}_i(m) &= \sum_{i \in \mathbb{I}} m(i) \\
m \sqcup m' &= \{(i, \mathsf{max}(m(i), m'(i))) \mid i \in \mathbb{I}\}
\end{aligned}$$

Fig. 2: A δ-CRDT counter; replica i.

and only returns that entry, instead of the full map as inc in the state-based CRDT counter does. This maintains the original semantics of the counter while allowing the smaller deltas returned by the delta-mutator to be sent, instead of the full map. As before, the received payload (whether one or more deltas) might not include entries for all keys in $\mathbb{I}$, which are assumed to have zero values. The decomposition is easy to understand in this example since the equation $\mathsf{inc}_i(X) = X \sqcup \mathsf{inc}_i^\delta(X)$ holds as $m\{i \mapsto m(i)+1\} = m \sqcup \{i \mapsto m(i)+1\}$. In other words, the single value for key i in the delta, corresponding to the local replica identifier, will overwrite the corresponding one in m since the former maps to a higher value (i.e., using max). Here it can be noticed that: (1) a delta *is* just a state, that can be joined possibly several times without requiring exactly-once delivery, and without being a representation of the "increment" operation (as in operation-based CRDTs), which is itself non-idempotent; (2) joining deltas into a delta-group and disseminating delta-groups at a lower rate than the operation rate reduces data communication overhead, since multiple increments from a given source can be collapsed into a single state counter.

5 Causal Consistency

Traditional state-based CRDTs converge using joins of the full state, which implicitly ensures per-object causal consistency [8]: each state of some replica of an object reflects the causal past of operations on the object (either applied locally, or applied at other replicas and transitively joined).

Therefore, it is desirable to have δ-CRDTs offer the same causal-consistency guarantees that standard state-based CRDTs offer. This raises the question about how can delta propagation and merging of δ-CRDT be constrained (and expressed in an anti-entropy algorithm) in such a manner to give the same results as if a standard state-based CRDT was used. Towards this objective, it is useful to define a particular kind of delta-group, which we call a *delta-interval*:

Definition 4 (Delta-interval). *Given a replica i progressing along the states $X_i^0, X_i^1, \ldots$, by joining delta d_i^k (either local delta-mutation or received delta-group) into X_i^k to obtain X_i^{k+1}, a delta-interval $\Delta_i^{a,b}$ is a delta-group resulting from joining deltas $d_i^a, \ldots, d_i^{b-1}$:*

$$\Delta_i^{a,b} = \bigsqcup\{d_i^k \mid a \leq k < b\}$$

The use of delta-intervals in anti-entropy algorithms will be a key ingredient towards achieving causal consistency. We now define a restricted kind of anti-entropy algorithm for δ-CRDTs.

Definition 5 (Delta-interval-based anti-entropy algorithm). *A given anti-entropy algorithm for δ-CRDTs is delta-interval-based, if all deltas sent to other replicas are delta-intervals.*

Moreover, to achieve causal consistency the next condition must satisfied:

Definition 6 (Causal delta-merging condition). *A delta-interval based anti-entropy algorithm is said to satisfy the causal delta-merging condition if the algorithm only joins $\Delta_j^{a,b}$ from replica j into state X_i of replica i that satisfy:*

$$X_i \sqsupseteq X_j^a.$$

This means that a delta-interval is only joined into states that at least reflect (i.e., subsume) the state into which the first delta in the interval was previously joined. The causal delta-merging condition is important since any delta-interval based anti-entropy algorithm of a δ-CRDT that satisfies it, can be used to obtain the same outcome of standard CRDTs; this is formally stated in Proposition 1.

Proposition 1. *(CRDT and δ-CRDT correspondence) Let (S, M, Q) be a standard state-based CRDT and (S, M^δ, Q) a corresponding delta-state decomposition. Any δ-CRDT state reachable by an execution E^δ over (S, M^δ, Q), by a delta-interval based anti-entropy algorithm A^δ satisfying the causal delta-merging condition, is equal to a state resulting from an execution E over (S, M, Q), having the corresponding data-type operations, by an anti-entropy algorithm A for state-based CRDTs.*

Proof. Please see the associated technical report [1].

Corollary 1. *(δ-CRDT causal consistency) Any δ-CRDT in which states are propagated and joined using a delta-interval-based anti-entropy algorithm satisfying the causal delta-merging condition ensures causal consistency.*

Proof. From Proposition 1 and causal consistency of state-based CRDTs.

5.1 Anti-Entropy Algorithm for Causal Consistency

Algorithm 1 is a delta-interval based anti-entropy algorithm which enforces the causal delta-merging condition. It can be used whenever the causal consistency guarantees of standard state-based CRDTs are needed. For simplicity, it excludes some optimizations that are important, but easy to derive, in practice. The algorithm distinguishes neighbor nodes, and only sends them delta-intervals that are joined at the receiving node, obeying the delta-merging condition.

Each node i keeps a contiguous sequence of deltas $d_i^l, \ldots, d_i^u$ in a map D from integers to deltas, with $l = \min(\mathsf{dom}(D))$ and $u = \mathsf{max}(\mathsf{dom}(D))$. The sequence numbers of deltas are obtained from the counter c_i that is incremented when a delta (whether a delta-mutation or delta-interval received) is joined with the current state. Each node i keeps an acknowledgments map A that stores, for each neighbor j, the largest index b for all delta-intervals $\Delta_i^{a,b}$ acknowledged by j (after j receives $\Delta_i^{a,b}$ from i and joins it into X_j).

Node i sends a delta-interval $d = \Delta_i^{a,b}$ with a (delta, d, b) message; the receiving node j, after joining $\Delta_i^{a,b}$ into its replica state, replies with an acknowledgment message (ack, b); if an ack from j was successfully received by node i,

```
inputs:
    $n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
durable state:
    $X_i \in S$, CRDT state; initially $X_i = \bot$
    $c_i \in \mathbb{N}$, sequence number; initially $c_i = 0$
volatile state:
    $D_i \in \mathbb{N} \hookrightarrow S$, sequence of deltas; initially $D_i = \{\}$
    $A_i \in \mathbb{I} \hookrightarrow \mathbb{N}$, acknowledges map; initially $A_i = \{\}$
on $\mathsf{receive}_{j,i}(\mathsf{delta}, d, n)$
    if $d \not\sqsubseteq X_i$ then
        $X_i' = X_i \sqcup d$
        $D_i' = D_i\{c_i \mapsto d\}$
        $c_i' = c_i + 1$
    $\mathsf{send}_{i,j}(\mathsf{ack}, n)$
on $\mathsf{receive}_{j,i}(\mathsf{ack}, n)$
    $A_i' = A_i\{j \mapsto \mathsf{max}(A_i(j), n)\}$
on $\mathsf{operation}_i(m^\delta)$
    $d = m^\delta(X_i)$
    $X_i' = X_i \sqcup d$
    $D_i' = D_i\{c_i \mapsto d\}$
    $c_i' = c_i + 1$
periodically // ship delta-interval or state
    $j = \mathsf{random}(n_i)$
    if $D_i = \{\} \vee \min(\mathsf{dom}(D_i)) > A_i(j)$ then
        $d = X_i$
    else
        $d = \bigsqcup\{D_i(l) \mid A_i(j) \leq l < c_i\}$
    if $A_i(j) < c_i$ then
        $\mathsf{send}_{i,j}(\mathsf{delta}, d, c_i)$
periodically // garbage collect deltas
    $l = \min\{n \mid (\_, n) \in A_i\}$
    $D_i' = \{(n, d) \in D_i \mid n \geq l\}$
```

Algorithm 1: Anti-entropy algorithm ensuring causal consistency of δ-CRDT.

it updates the entry of j in the acknowledgment map, using the max function. This handles possible old duplicates and messages arriving out of order.

Like the δ-CRDT state, the counter c_i is also kept in a durable storage. This is essential to avoid conflicts after potential crash and recovery incidents. Otherwise, there would be the danger of receiving some delayed ack, for a delta-interval sent before crashing, causing the node to skip sending some deltas generated after recovery, thus violating the delta-merging condition.

The algorithm for node i periodically picks a random neighbor j. In principle, i sends the join of all deltas starting from the latest delta acked by j and forward. Exceptionally, i sends the entire state in two cases: (1) if the sequence of deltas D_i is empty, or (2) if j is expecting from i a delta that was already removed from D_i (e.g., after a crash and recovery, when both deltas and the ack map, being volatile state, are lost); i tracks this in $A_i(j)$. To garbage collect old deltas, the algorithm periodically removes the deltas that have been acked by *all* neighbors.

Proposition 2. *Algorithm 1 produces the same reachable states as a standard algorithm over a CRDT for which the δ-CRDT is a decomposition.*

Proof. Please see the associated technical report [1].

6 δ-CRDTs for Add-Wins OR-Sets

An Add-wins Observed-Remove Set (OR-set) is a well-known CRDT datatype that offers the same sequential semantics of a sequential set and adopts a specific

$$\begin{aligned}
\Sigma &= \mathcal{P}(\mathbb{I} \times \mathbb{N} \times E) \times \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\sigma_i^0 &= (\{\}, \{\}) \\
\mathsf{add}_i^\delta(e, (s, t)) &= (\{(i, n+1, e)\}, \{\}) \\
&\quad \text{with } n = \mathsf{max}(\{k \mid (i, k, \_) \in s\}) \\
\mathsf{rmv}_i^\delta(e, (s, t)) &= (\{\}, \{(j, n) \mid (j, n, e) \in s\}) \\
\mathsf{elements}_i((s, t)) &= \{e \mid (j, n, e) \in s \wedge (j, n) \notin t\} \\
(s, t) \sqcup (s', t') &= (s \cup s', t \cup t')
\end{aligned}$$

(a) With Tombstones

$$\begin{aligned}
\Sigma &= \mathcal{P}(\mathbb{I} \times \mathbb{N} \times E) \times \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\sigma_i^0 &= (\{\}, \{\}) \\
\mathsf{add}_i^\delta(e, (s, c)) &= (\{(i, n+1, e)\}, \{(i, n+1)\}) \\
&\quad \text{with } n = \mathsf{max}(\{k \mid (i, k) \in c\}) \\
\mathsf{rmv}_i^\delta(e, (s, c)) &= (\{\}, \{(j, n) \mid (j, n, e) \in s\}) \\
\mathsf{elements}_i((s, c)) &= \{e \mid (j, n, e) \in s\} \\
(s, c) \sqcup (s', c') &= ((s \cap s') \cup \{(i, n, e) \in s \mid (i, n) \notin c'\} \\
&\quad \cup \{(i, n, e) \in s' \mid (i, n) \notin c\}, c \cup c')
\end{aligned}$$

(b) Without Tombstones (optimized)

Fig. 3: Add-wins observed-remove δ-CRDT set, replica i.

resolution semantics for operations that concurrently add and remove the same element. Add-wins means that an add prevails over a concurrent remove. Remove operations, however, only affect elements added by causally preceding adds. The purpose of these δ-CRDT OR-set versions is to design δ-mutators that return small deltas to be lightly disseminated, as discussed above, instead of shipping the entire state as in classical CRDTs [22,23,5].

6.1 Add-wins OR-Set with tombstones

Fig. 3a depicts a simple, but inefficient, δ-CRDT implementation of a state-based add-wins OR-Set. The state Σ consists of a set of tagged elements and a set of tags, acting as tombstones. Globally unique tags of the form $\mathbb{I} \times \mathbb{N}$ are used and ensured by pairing a replica identifier in $\mathbb{I}$ with a monotonically increasing natural counter. Once an element $e \in E$ is added to the set, the delta-mutator add^δ creates a globally unique tag by incrementing the highest tag present in its local state and that was created by replica i itself (max returns 0 if no tag is present). This tag is paired with value e and stored as a new unique triple in s. Since an "add" wins any concurrent "remove", removing an element e should only be tombstoned if it was preceded by an add operation (i.e., the element is in s), otherwise it has no effect. Consequently, the delta-mutator rmv^δ retains in the tombstone set all tags associated to element e, being removed from the local state. This is essential to prevent a removed element to reappear once the local state is merged with another replica state that still have that element (i.e., it has not been removed yet remotely as replicas are loosely coupled). The function $\mathsf{elements}$ returns only the elements that are added but not yet tombstoned. Join $\sqcup$ simply unions the respective sets that are, therefore, both grow-only.

6.2 Optimized Add-wins OR-Set

A more efficient design is presented in Fig. 3b allowing also the set of tagged elements (i.e., tombstone set above) to shrink as elements are removed. This design offers the same semantics and have a similar state structure to the former; however, it has a different behavior. Now, $\mathsf{elements}$ returns all the elements in the tagged set s, without consulting t as before. Added and removed items are now tagged in the *causal context set* c. Although, the set c and t look similar in structure, they have a different behavior (we call it c instead of t to remove this confusion): a tombstone set t simply stores all removed elements tags, while c retains only the causal information needed to add/remove an element. For presentation simplicity, c in Fig. 3b simply retains all removed elements tags; however, after compression, c will be very concise and look different from t; this is explained in the next section.

Adding an element creates a unique tag by resorting to the causal context c (instead of s). The tag is paired with the element and added to s (as before). The difference is that the new tag is also added to the causal context set c. The delta-mutator rmv^δ is the same as before, adding all tags associated to the element being removed to c. The desired semantics are maintained by the novel join operation $\sqcup$. To join two states, their causal contexts c are simply unioned; whereas, the new element set s only preserves: (1) the triples present in both sets (therefore, not removed in either), and also (2) any triple present in one of the sets and whose tag is not present in the causal context of the other state.

Causal Context Compression In practice, the causal context c can be efficiently compressed without any loss of information. When using an anti-entropy algorithm that provides causal consistency, e.g., Algorithm 1, then for each replica state $X_i = (s_i, c_i)$ and replica id $j \in \mathbb{I}$, we have a contiguous sequence:

$$1 \leq n \leq \mathsf{max}(\{k \mid (j,k) \in c_i\}) \Rightarrow (j,n) \in c_i.$$

Thus, the causal context can always be encoded as a compact version vector [21] $\mathbb{I} \hookrightarrow \mathbb{N}$ that keeps the maximum sequence number for each replica. Even under non-causal anti-entropy, compression is still possible by keeping a version vector that encodes the offset of the contiguous sequence of tags from each replica, together with a set for the non-contiguous tags. As anti-entropy proceeds, each tag is eventually encoded in the vector, and thus the set remains typically small. Compression is less likely for the causal context of delta-groups in transit or buffered to be sent, but those contexts are only transient and smaller than those in the actual replica states. Moreover, the same techniques that encode contiguous sequences of tags can also be used for transient context compression [19].

7 Optimized Multi-value Register δ-CRDT

Multi-Value Registers (MVR) are popular constructions in which a read operation returns the set of values concurrently written, but not causally overwritten; these values are then reduced to a single value by applications [13]. Until now, these types have been implemented by assigning a version vector to each written value [22,8]. In Figure 4, we show that the optimization that was developed

$$
\begin{aligned}
\Sigma &= \mathcal{P}(\mathbb{I} \times \mathbb{N} \times V) \times \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\sigma_i^0 &= (\{\}, \{\}) \\
\mathsf{wr}_i^\delta(v, (s, c)) &= (\{(i, n+1, v)\}, \{(i, n+1)\} \cup \{(j, m) \mid (j, m, \_) \in s\}) \text{ with } n = \mathsf{max}(\{k \mid (i, k) \in c\}) \\
\mathsf{rd}_i((s, c)) &= \{v \mid (j, n, v) \in s\} \\
(s, c) \sqcup (s', c') &= ((s \cap s') \cup \{(i, n, v) \in s \mid (i, n) \notin c'\} \cup \{(i, n, v) \in s' \mid (i, n) \notin c\}, c \cup c')
\end{aligned}
$$

Fig. 4: Optimized δ-CRDT multi-value register, replica i.

for Sets, can also be used to compactly tag the values in a multi-value register. On a write operation wr, it is enough to assign a new scalar tag, from $\mathbb{I} \times \mathbb{N}$, using a replica id i and counter to uniquely tag the written value v. To ensure that values overwritten are deleted, the produced causal context c lists all tags associated to those values. Since those values are absent from the payload set s they will be deleted in replicas that still have them, applying join definition $\sqcup$ (that is in common with Figure 3b). The causal context compression techniques defined earlier also apply here.

8 Message Complexity

Our delta-based framework, δ-CRDT, clearly introduces significant cost improvements on messaging. Despite being a generic framework, δ-CRDT requires delta mutators to be defined per datatype. This makes the bit-message complexity datatype-based rather than generic. To give an intuition about this complexity, we address the three datatypes introduced above: *counter*, OR-Set, and MVR.

Counters. In classical state-based CRDTs, the entire map of a *counter* is shipped. As the map-size grows with the number of replicas, this leads a bit-message complexity of $\widetilde{O}(|\mathbb{I}|)$ [1]. In the δ-CRDT case, only recently updated map entries α are shipped yielding a bit-complexity $\widetilde{O}(\alpha)$, where $\alpha \ll |\mathbb{I}|$.

OR-set. Shipping in classical OR-set CRDTs delivers the entire state which yields a bit-message complexity of $O(S)$, where S is the state-size. In δ-CRDT, only deltas are shipped, which renders a bit-message complexity $O(s)$ where s represents the size of the recent updates occurred since the last shipping. Clearly, $s \ll S$ since the updates that occur on a state in a period of time are often much less than the total number of items.

MVR. In classical MVR, the worst case state is composed of $|\mathbb{I}|$ concurrently written values, each associated with a $|\mathbb{I}|$ sized version vector. This makes the bit-message complexity $\widetilde{O}(|\mathbb{I}|^2)$. In the novel delta design in Figure 4, no version vector is used, whereas the number of possible values remain the same (summing up the values set s and meta-data in c), this reduces the bit-message complexity to $\widetilde{O}(|\mathbb{I}|)$ as well as the worst case state complexity.

[1] $\widetilde{O}$ is a variant of big O ignoring logarithmic factors in the size of integers and ids.

9 Related Work

Eventually convergent data types. The design of replicated systems that are always available and eventually converge can be traced back to historical designs in [25,16], among others. More recently, replicated data types that always eventually converge, both by reliably broadcasting operations (called operation-based) or gossiping and merging states (called state-based), have been formalized as CRDTs [17,4,22,23]. These are also closely related to Bloom^L [10] and Cloud Types [7].

Deltas. A key feature of δ-CRDT is message size reduction (not improving local state lower bounds [8]), by using small-sized deltas, while preserving the advantages of classical state-based CRDTs. The general old idea of using differences between things, called "deltas" in many contexts, can lead to many designs, depending on how exactly a delta is defined. The state-based deltas introduced for Computational CRDTs [20] require an extra delta-specific merge (in addition to the standard join) which does not ensure idempotence. In [14], an improved synchronization method for non-optimized OR-set CRDT [22] is presented, where delta information is propagated; in that paper deltas are a collection of items (related to update events between synchronizations), manipulated and merged through a protocol, as opposed to normal states in the semilattice. No generic framework is defined (that could encompass other data types) and the protocol requires several communication steps to compute the information to exchange.

Operation-based CRDTs. These CRDTs [22,23,3] also support small message sizes, and in particular, *pure* flavors [3] that restrict messages to the operation name, and possible arguments. Though pure operation-based CRDTs allow for compact states and are very fast at the source (since operations are broadcast without consulting the local state), the model requires more systems guarantees than δ-CRDT do, e.g., exactly-once reliable delivery and membership information, and impose more complex integration of new replicas. The work in [9] shows a different trade-off among state deltas and pure operations, by tagging operations and creating a globally stable log of operations while allowing local transient logs to preserve availability. While having other advantages, the creation of this global log requires more coordination than our gossip approach for causally consistent delta dissemination, and can stall dissemination.

Encoding causal histories. State-based CRDT are always designed to be causally consistent [4,23]. Optimized implementations of sets, maps, and multi-value registers can build on this assumption to keep the meta-data small [8]. In δ-CRDT, however, deltas and delta-groups are normally not causally consistent, and thus the design of *join*, the meta-data state, as well as the anti-entropy algorithm used must ensure this. Without causal consistency, the causal context in δ-CRDT can not always be summarized with version vectors, and consequently, techniques that allow for gaps are often used. A well known mechanism that allows for encoding of gaps is found in Concise Version Vectors [18]. Interval Version Vectors [19], later on, introduced an encoding that optimizes sequences and allows gaps, while preserving efficiency when gaps are absent.

10 Conclusion

We introduced the new concept of δ-CRDTs and devised *delta-mutators* over state-based datatypes which can detach the changes that an operation induces on the state. This brings a significant performance gain as it allows only shipping small states, i.e., *deltas*, instead of the entire state. The significant property in δ-CRDT is that it preserves the crucial properties (idempotence, associativity and commutativity) of standard state-based CRDT. In the worst case, deltas can be forgotten and the entire state can always be shipped, allowing scenarios such as long duration partitions, which would be problematic for op-based CRDTs.

In addition, we have shown how δ-CRDT can achieve causal consistency; and we presented an anti-entropy algorithm that allows replacing classical state-based CRDTs by more efficient ones, while preserving their properties. As an application for our approach, we designed two novel δ-CRDT specifications for two well-known datatypes: an optimized observed-remove set [5] and an optimized multi-value register [13].

References

1. Almeida, P.S., Shoker, A., Baquero, C.: Efficient state-based crdts by delta-mutation. CoRR abs/1410.2803 (2014), http://arxiv.org/abs/1410.2803
2. Baquero, C.: Delta-enabled-crdts, github.com/CBaquero/delta-enabled-crdts
3. Baquero, C., Almeida, P.S., Shoker, A.: Making operation-based CRDTs operation-based. In: Proceedings of Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference. Springer (2014)
4. Baquero, C., Moura, F.: Using structural characteristics for autonomous operation. Operating Systems Review 33(4), 90–96 (1999)
5. Bieniusa, A., Zawirski, M., Preguiça, N., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: An optimized conflict-free replicated set. Rapp. Rech. RR-8083, INRIA, Rocquencourt, France (Oct 2012), http://hal.inria.fr/hal-00738680
6. Brown, R., Cribbs, S., Meiklejohn, C., Elliott, S.: Riak dt map: A composable, convergent replicated dictionary. In: Proceedings of the First Workshop on Principles and Practice of Eventual Consistency. pp. 1:1–1:1. PaPEC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2596631.2596633
7. Burckhardt, S., Fähndrich, M., Leijen, D., Wood, B.P.: Cloud types for eventual consistency. In: ECOOP 2012–Object-Oriented Programming, pp. 283–307. Springer (2012)
8. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: Jagannathan, S., Sewell, P. (eds.) POPL. pp. 271–284. ACM (2014)
9. Burckhardt, S., Leijen, D., Fahndrich, M.: Cloud types: Robust abstractions for replicated shared state. Tech. Rep. MSR-TR-2014-43 (March 2014), http://research.microsoft.com/apps/pubs/default.aspx?id=211340

10. Conway, N., Marczak, W.R., Alvaro, P., Hellerstein, J.M., Maier, D.: Logic and lattices for distributed programming. In: Proceedings of the Third ACM Symposium on Cloud Computing. p. 1. ACM (2012)
11. Cribbs, S., Brown, R.: Data structures in Riak. In: Riak Conference (RICON). San Francisco, CA, USA (oct 2012)
12. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order (2. ed.). Cambridge University Press (2002)
13. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles (SOSP). Operating Systems Review, vol. 41, pp. 205–220. Assoc. for Computing Machinery, Stevenson, Washington, USA (Oct 2007)
14. Deftu, A., Griebsch, J.: A scalable conflict-free replicated set data type. In: Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems. pp. 186–195. ICDCS '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/ICDCS.2013.10
15. Helland, P.: Idempotence is not a medical condition. Queue 10(4), 30:30–30:46 (Apr 2012), http://doi.acm.org/10.1145/2181796.2187821
16. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (Jan 1976), http://www.rfc-editor.org/rfc.html
17. Letia, M., Preguiça, N., Shapiro, M.: CRDTs: Consistency without concurrency control. Rapp. Rech. RR-6956, INRIA, Rocquencourt, France (Jun 2009), http://hal.inria.fr/inria-00397981/
18. Malkhi, D., Terry, D.: Concise version vectors in winfs. Distributed Computing 20(3), 209–219 (2007)
19. Mukund, M., R., G.S., Suresh, S.P.: Optimized or-sets without ordering constraints. In: Proceedings ot the International Conference on Distributed Computing and Networking. p. 227241. ACM, New York, NY, USA (2014)
20. Navalho, D., Duarte, S., Preguiça, N., Shapiro, M.: Incremental stream processing using computational conflict-free replicated data types. In: Proceedings of the 3rd International Workshop on Cloud Data and Platforms. pp. 31–36. ACM (2013)
21. Parker, D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. IEEE Trans. Softw. Eng. 9(3), 240–247 (May 1983), http://dx.doi.org/10.1109/TSE.1983.236733
22. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, INRIA, Rocquencourt, France (Jan 2011), http://hal.archives-ouvertes.fr/inria-00555588/
23. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS). Lecture Notes in Comp. Sc., vol. 6976, pp. 386–400. Springer-Verlag, Grenoble, France (Oct 2011)
24. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Symp. on Op. Sys. Principles (SOSP). pp. 172–182. ACM SIGOPS, ACM Press, Copper Mountain, CO, USA (Dec 1995)
25. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC). pp. 233–242. Vancouver, BC, Canada (Aug 1984)

B.2 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, Marc Shapiro. Putting Consistency Back into Eventual Consistency. In Proc. EuroSys'15.

Putting Consistency Back into Eventual Consistency

Valter Balegas Sérgio Duarte Carla Ferreira Rodrigo Rodrigues Nuno Preguiça
NOVA LINCS, FCT, Universidade Nova Lisboa

Mahsa Najafzadeh Marc Shapiro
Inria Paris-Rocquencourt & Sorbonne Universités, UPMC Univ Paris 06, LIP6

Abstract

Geo-replicated storage systems are at the core of current Internet services. The designers of the replication protocols used by these systems must choose between either supporting low-latency, eventually-consistent operations, or ensuring strong consistency to ease application correctness. We propose an alternative consistency model, *Explicit Consistency*, that strengthens eventual consistency with a guarantee to preserve specific invariants defined by the applications. Given these application-specific invariants, a system that supports Explicit Consistency identifies which operations would be unsafe under concurrent execution, and allows programmers to select either violation-avoidance or invariant-repair techniques. We show how to achieve the former, while allowing operations to complete locally in the common case, by relying on a *reservation* system that moves coordination off the critical path of operation execution. The latter, in turn, allows operations to execute without restriction, and restore invariants by applying a repair operation to the database state. We present the design and evaluation of Indigo, a middleware that provides Explicit Consistency on top of a causally-consistent data store. Indigo guarantees strong application invariants while providing similar latency to an eventually-consistent system in the common case.

1. Introduction

To improve user experience in services that operate on a global scale, from social networks and multi-player online games to e-commerce applications, the infrastructure that supports these services often resorts to geo-replication [9, 10, 12, 25, 27, 28, 41], i.e., maintains copies of application data and logic in multiple datacenters scattered across the globe. This ensures low latency, by routing requests to the closest datacenter, but only when the request does not require cross-datacenter synchronization. Executing update operations without cross-datacenter synchronization is normally achieved through weak consistency. The downside of weak consistency models is that applications have to deal with concurrent operations, which can lead to non-intuitive and undesirable semantics.

These semantic anomalies do not occur in systems that enforce strict serializability, i.e., serialize all operations in real-time order. Weaker models, such as serializability or snapshot isolation, relax synchronization, but still require frequent coordination among replicas, which increases latency and decreases availability. A promising alternative is to try to combine the strengths of both approaches by supporting both weak and strong consistency, depending on the operation [25, 41, 43]. In this approach, operations requiring strong consistency still incur high latency and are unavailable when the system partitions. Additionally, these systems make it harder to design applications, as operations need to be correctly classified to guarantee the correctness of the application.

In this paper, we propose *Explicit Consistency* as an alternative consistency model, in which an application specifies the invariants, or consistency rules, that the system must maintain. Unlike models defined in terms of execution orders, Explicit Consistency is defined in terms of application properties: the system is free to reorder execution of operations at different replicas, provided that the specified invariants are maintained.

In addition, we show that it is possible to implement explicit consistency while mostly avoiding cross-datacenter coordination, even for critical operations that could potentially break invariants. To this end, we propose a three-step methodology to derive a safe version of the application. First, we use static analysis to infer which operations can be safely executed without coordination. Second, for the remaining operations, we provide the programmer with a choice of invariant-repair [38] or violation-avoidance techniques. Finally, application code is instrumented with the appropriate calls to our middleware library.

Violation-avoidance extends escrow and reservation approaches [15, 17, 32, 35, 39]. The idea is that a replica coordinates in advance, to pre-allocate the permission to execute some collection of future updates, which (thanks to the reser-

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2741948.2741972

vation) will require no coordination. This amortizes the cost and moves it off the critical path.

Finally, we present the design of Indigo, a middleware for Explicit Consistency built on top of a geo-replicated key-value store. Indigo is designed in a way that is agnostic to the details of the underlying key-value store, only requiring it to ensure properties that are known to be efficient to implement, namely per-key, per-replica linearizability, causal consistency, and transactions with weak semantics [2, 27, 28].

In summary, we make the following contributions:

- Explicit Consistency, a new consistency model for application correctness, centered on the application semantics, and not on the order of operations.
- A methodology to derive an efficient reservation system for enforcing Explicit Consistency, based on the set of invariants associated with the application.
- Indigo, a middleware system implementing Explicit Consistency on top of a causally consistent geo-replicated key-value store.

The remainder of the paper is organized as follows: Section 2 introduces Explicit Consistency. Section 3 gives an overview of our approach. Section 4 presents the analysis for detecting unsafe concurrent operations. Section 5 details the techniques for handling these operations. Section 6 discusses the implementation of Indigo and Section 7 presents an evaluation of the system. Related work is discussed in Section 8. Finally, Section 9 concludes the paper.

2. Explicit Consistency

In this section we define precisely the consistency guarantees that Indigo provides. We start by defining the system model, and then how Explicit Consistency restricts the set of behaviors allowed by that model.

To illustrate the concepts, we use as running example the management of tournaments in a distributed multi-player game. The game maintains information about players and tournaments. Players can register and de-register from the game. Players compete in tournaments, for which they can enroll and disenroll. A set of matches occurs for each tournament. Each tournament has a maximum capacity. In some cases, e.g., when there are not enough participants, a tournament can be canceled before it starts. Otherwise a tournament's life cycle is creation, start, and end.

2.1 System Model and Definitions

We consider a database composed of a set of objects in a typical cloud deployment, where data is fully replicated in multiple datacenters, and partitioned inside each datacenter.

Applications access and modify the database by issuing high-level operations. These operations consist of a sequence of *read* and *write* operations enclosed in *transactions*. An application submits a transaction to a replica; its reads and writes execute on a private copy of the replica state. If the transaction commits, its writes are applied to the local replica (local transaction), and propagate asynchronously to remote replicas, where they are also applied (remote transaction). If the transaction aborts, it has no effect.

We denote by $t(S)$ the state after applying the write operations of committed transaction t to some state S. We define a database snapshot, S_n, as the state of the database after a sequence of committed transactions $t_1, \ldots, t_n$ from the initial database state, S_{init}, i.e., $S_n = t_n(\ldots(t_1(S_{init})))$. The state of a replica results from applying both local and remote transactions, in the order received.

The transaction set $T(S)$ of a database snapshot S is the set of transactions included in S, e.g., $T(S_n) = \{t_1, \ldots, t_n\}$. We say that a transaction t_a executing in a database snapshot S_a happened-before t_b executing in S_b, $t_a \prec t_b$, if $t_a \in T(S_b)$. Two transactions t_a and t_b are concurrent, $t_a \parallel t_b$, if $t_a \not\prec t_b \wedge t_b \not\prec t_a$ [24].

For a given set of transactions T, the happens-before relation defines a partial order among them, $O = (T, \prec)$. We say $O' = (T, <)$ is a valid serialization of $O = (T, \prec)$ if O' is a linear extension of O, i.e., $<$ is a total order compatible with $\prec$.

Transactions can execute concurrently, with each replica executing transactions according to a different valid serialization. We assume the system guarantees state convergence, i.e., all valid serializations of $(T, \prec)$ lead to the same database state. Different techniques can be used to this end, from a simple *last-writer-wins* strategy to more complex approaches based on conflict-free replicated data types (CRDTs) [38, 41].

2.2 Explicit Consistency

Explicit Consistency is a novel consistency semantics for replicated systems, where programmers define the application-specific correctness rules that should be met. These rules are expressed as invariants over the database state.

Even if each replica maintains some invariant locally, concurrent updates might still cause violation. Consider for instance a tournament with a maximum capacity, limiting the cardinality of the set of enrolled players. Two replicas could concurrently enroll players into the same tournament, each one respecting the capacity. However, if the merge function is the union of the two sets of players, the capacity might be exceeded nonetheless.

Our formal definition starts with the helper definition of an invariant I, as a logical condition over the state of the database. We say that state S is an *I-valid state* if I holds in S, i.e., if $I(S) = true$.

Definition 2.1 (I-valid serialization)**.** Given a set of transactions T and its associated happens-before partial order $\prec$, $O_i = (T, <)$ is an *I-valid serialization* of $O = (T, \prec)$ if O_i is a valid serialization of O, and I holds in every state that results from executing some prefix of O_i.

We can now formally define the conditions that a system must uphold to ensure Explicit Consistency.

Definition 2.2 (Explicit consistency)**.** A system provides Explicit Consistency if all serializations of $O = (T, \prec)$ are I-valid serializations, where T is the set of transactions executed in the system and $\prec$ their associated partial order.

This concept is related to the *I-confluence* of Bailis et al. [5]. I-confluence defines the conditions under which operations may execute concurrently, while still ensuring that the system converges to an I-valid state. The current work generalizes this to cases where coordination is needed, and furthermore proposes efficient solutions.

3. Overview

Given application invariants, our approach for Explicit Consistency has three steps: *(i)* Detect the sets of operations that may lead to invariant violation when executed concurrently, called *I-offender sets*. *(ii)* Select an efficient mechanism for handling I-offender sets. *(iii)* Instrument the application code to use the selected mechanism on top of a weakly consistent database system.

The first step consists of discovering I-offender sets. This analysis is based on a model of the effects of operations. This information is provided by the application programmer, as annotations specifying the changes performed by each operation. Using this information, combined with the application invariants, static analysis infers the sets of operation invocations that, when executed concurrently, may lead to invariant violation (I-offender sets). Conceptually, the analysis considers all reachable database states and, for each state, all sets of operation invocations that can execute in that state; it checks if executing these operations concurrently might cause an invariant violation. Obviously, it is not feasible to exhaustively consider all database states and operation sets; instead, a practical approach is to use static verification techniques, which are detailed in Section 4.

In the second, the developer decides which approach to use to handle the I-offender sets. There are two options. With the first, *invariant repair*, operations are allowed to execute concurrently, and the conflict resolution rules that merge their outputs should include code to restore the invariants. One example is a graph data structure that supports operations to add and remove vertices and edges; if one replica adds an edge while concurrently another replica removes one of the edge's vertices, the merged state might ignore the hanging edge to ensure the invariant that an edge connects two vertices [38]. A similar approach applies to trees [30].

The second option, *violation avoidance*, consists of restricting concurrency sufficiently to avoid the invariant violation. We propose a number of techniques to allow a replica to execute such operations safely, without coordinating frequently with the others. Consider for instance the enrollment invariant (if a player is enrolled in a tournament, both the player and the tournament must exist). Any replica is allowed to execute the *enrollTournament* operation without coordination, as long as all replicas are forbidden to run *removePlayer* and *removeTournament*. This *reservation* may apply to a particular subset of players and tournaments.

Our reservation mechanisms support such functionality with reservations tailored to the different types of invariants, as detailed in Section 5.

In the third step, the application code is instrumented to use the conflict-repair and conflict-avoidance mechanisms selected by the programmer. This involves extending operations to call the appropriate API functions supported by Indigo.

4. Determining I-offender sets

In this section we detail the first step of our approach.

4.1 Defining invariants and post-conditions

Defining application invariants An application invariant is described by a first-order logic formula. More formally, we assume the invariant is an universally quantified formula in prenex normal form[1]

$$\forall x_1, \cdots, x_n, \varphi(x_1, \cdots, x_n).$$

First-order logic formulas can express a wide variety of consistency constraints; we give some examples in Section 4.2.

An invariant can use predicates such as $player(P)$ or $enrolled(P, T)$. A user may interpret them to mean that P is a player and that P is enrolled in tournament T; but technically the system depends only on their truth values and on the formulas that relate them. The application developer needs only to specify the effects of operations on the truth values of the terms used in the invariant.

Similarly, numeric restrictions can be expressed through the use of functions. For example, we may use $nrPlayers(T)$ (the number of players in tournament T) to limit the size of a tournament: $\forall T, nrPlayers(T) \leq 5$.

An application must satisfy the conjunction of all invariants.

Defining operation postconditions To express the side effects of operations, postconditions state the properties that are ensured after the execution of an operation that modifies the database. There are two types of side effect clauses: predicate clauses, which describe a truth assignment for a predicate (stating whether the predicate is true or false after execution of the operation); and function clauses, which define the relation between the initial and final function values. To give some examples, operation $removePlayer(P)$, which removes player P, has a postcondition with predicate clause $\neg player(P)$, stating that predicate $player$ is false for player P. Operation $enrollTournament(P, T)$,

[1] Formula $\forall x, \varphi(x)$ is in prenex normal form if clause φ is quantifier-free. Every first-order logic formula has an equivalent prenex normal form.

which enrolls player P into tournament T, has a postcondition with two clauses, $\mathit{enrolled}(P, T)$ and $\mathit{nrPlayers}(T) = \mathit{nrPlayers}(T) + 1$. If the player is already enrolled, the operation produces no side effects.

The syntax for postconditions is given by the grammar:

$$\begin{array}{lcl}
\mathit{post} & ::= & \mathit{clause}_1 \wedge \mathit{clause}_2 \wedge \cdots \wedge \mathit{clause}_k \\
\mathit{clause} & ::= & \mathit{pclause} \mid \mathit{fclause} \\
\mathit{pclause} & ::= & p(o_1, o_2, \cdots, o_n) \mid \neg p(o_1, o_2, \cdots, o_n) \\
\mathit{fclause} & ::= & f(o_1, o_2, \cdots, o_n) = \mathit{opr} \mid \mathit{opr} \oplus \mathit{opr} \\
\mathit{opr} & ::= & n \mid f(o_1, o_2, \cdots, o_n) \\
\oplus & ::= & + \mid - \mid * \mid \ldots
\end{array}$$

where p and f are predicates and functions respectively, over objects $o_1, o_2, \cdots, o_n$.

Although we impose that a postcondition is a conjunction, it is possible to deal with operations that have alternative side effects, by splitting the alternatives between multiple dummy operations. For example, an operation φ with postcondition $\varphi_1 \vee \varphi_2$ could be replaced by operations op_1 and op_2 with postconditions φ_1 and φ_2, respectively.

4.2 Expressiveness of Application Invariants

Despite the simplicity of our model, it can express significant classes of invariants, as discussed next.

4.2.1 Restrictions Over The State

An application can define the set of valid application states, using invariants that define conditions that must be satisfied in every database state. By combining user-defined predicates and functions, it is possible to address a wide range of application semantics.

Numeric constraints Numeric constraints refer to numeric properties of the application and set lower or upper bounds to data values. Often, they control the use or access to a limited resource. For example, to ensure that a player does not overspend her (virtual) budget: $\forall P, \mathit{player}(P) \Rightarrow \mathit{budget}(P) \geq 0$. Disallowing an experienced player from participating in a beginner tournament can be expressed as: $\forall T, P, \mathit{enrolled}(P, T) \wedge \mathit{beginners}(T) \Rightarrow \mathit{score}(P) \leq 30$. By using user-defined functions in the constraints, it is possible to express complex conditions over the database state. We have previously shown how to limit the number of enrolled players in a tournament by using a function that counts the enrolled players. The same approach can be used to limit the number of elements in the database that satisfy any generic condition.

Uniqueness, a common correctness property, may also be expressed using a counter function. For example, the formula $\forall P, \mathit{player}(P) \Rightarrow \mathit{nrPlayerId}(P) = 1$, states that P must have a unique player identifier. Whereas, the formula $\forall T, \mathit{tournament}(T) \Rightarrow \mathit{nrLeaders}(T) = 1$ states that a collection has exactly one leader.

Integrity constraints An integrity constraint specifies the relationships between different objects, such as the *foreign key constraint* in databases. A typical example is the one at the beginning of this section, stating that enrollment must refer to existing players and tournaments. If the tournament application had a score table for players, another integrity constraint might be that every table entry must belong to an existing player: $\forall P, \mathit{hasScore}(P) \Rightarrow \mathit{player}(P)$.

General constraints over the state An invariant may also capture general constraints. For example, consider an application to reserve meetings, where two meetings must not overlap in time. Using predicate $\mathit{time}(M, S, E)$ to mean that meeting M starts at time S and ends at time E, we could write this invariant as follows: $\forall M_1, M_2, S_1, S_2, E_1, E_2, \mathit{time}(M_1, S_1, E_1) \wedge \mathit{time}(M_2, S_2, E_2) \wedge M_1 \neq M_2 \Rightarrow E_2 \leq S_1 \vee S_2 \geq E_1$.

4.2.2 Restrictions Over State Transitions

In addition to conditions over database state, we support some forms of temporal specifications, i.e., restrictions over state transitions. Our approach is to turn this into an invariant over the state of the database, by augmenting the database with a so-called *history variable* that records its state in a given moment in the past [1, 33].

In our running example, we might want to state, for instance, that players may not enroll or drop from an active tournament, i.e., between the start and the end of the tournament. For this, when a tournament starts, the application stores the list of participants, which can later be checked against the list of enrollments. If $\mathit{participant}(P, T)$ asserts that player P participates in active tournament T, and $\mathit{active}(T)$ asserts that tournament T is active, the above rule can be specified as follows: $\forall P, T, \mathit{active}(T) \wedge \mathit{enrolled}(P, T) \Rightarrow \mathit{participant}(P, T)$.

An alternative is to use a logic with support for temporal logic expressions, which allow for writing expressions that specify rules over time [24, 34]. Such approach would require more complex specification for programmers and a more complex analysis. We decided to forgo temporal logic, since our experience showed that our simpler approach was sufficient for specifying common application invariants.

4.2.3 Existential quantifiers

Some properties require existential quantifiers, for instance to state that tournaments must have at least one player enrolled: $\forall T, \mathit{tournament}(T) \Rightarrow \exists P, \mathit{enrolled}(P, T)$. This can be easily handled, since the existential quantifier can be replaced by a function, using the technique called skolemization. For this example, we may use function $\mathit{nrPlayers}(T)$ as such: $\forall T, \mathit{tournament}(T) \Rightarrow \mathit{nrPlayers}(T) \geq 1$.

4.2.4 Uninterpreted predicates and functions

The fact that predicates and functions are uninterpreted imposes limitations to the invariants that can be expressed. It implies, for example, that it is not possible to express reachability properties or other properties over recursive data structures. To encode invariants that require such properties, the

```
@Invariant("forall(P : p, T : t) :- enrolled(p, t) =>
player(p) and tournament(t)")
@Invariant("forall(P : p) :- budget(p) >= 0")
@Invariant("forall(T : t) :- nrPlayers(t) <= Capacity")
@Invariant("forall(T : t) :- active(t)
=> nrPlayers(t) >= 1")
@Invariant("forall(T : t, P : p) :- active(t) and
enrolled(p,t) => participant(p, t)")
public interface ITournament {
 @True("player($0)")
 void addPlayer(P p);

 @False("player($0)")
 void removePlayer(P p);

 @True("tournament($0)")
 void addTournament(T t);

 @False("tournament($0)")
 void removeTournament(T t);

 @True("enrolled($0, $1)")
 @False("participant($0, $1)")
 @Increments("nrPlayers($1, 1)")
 @Decrements("budget($0, 1)")
 void enrollTournament(P p, T t);

 @False("enrolled($0, $1)")
 @Decrements("nrPlayers($1, 1)")
 void disenrollTournament(P p, T t);

 @True("active($0)")
 @True("participant(_, $0)")
 void beginTournament(T t);

 @False("active($0)")
 void endTournament(T t);

 @Increments("budget($0,$1)")
 void addFunds(P p, int amount);
}
```

Figure 1. Invariant specification for the tournament application in Java (excerpt)

programmer has to express predicates that encode coarser statements over the database, which lead to a conservative view of safe concurrency. For example, instead of specifying some property over a branch of a tree, the programmer can define the property over the whole tree.

4.2.5 Example

In Figure 1 shows how to express the invariants for the tournament application in our Java prototype. The invariants in the listing are a subset of the examples just discussed. Application invariants are entered as Java annotations to the application interface (or class), and operation side-effects as annotations to the corresponding methods. Our notation was defined to be simple to convert to the language of the Z3 theorem prover, used in our prototype.

4.3 Algorithm

To identify the sets of concurrent operations that may lead to an invariant violation, we perform static analysis of operation postconditions against invariants. This analysis focuses on the case where operations execute concurrently from the same state. Although we assume that in a sequential execution, the invariants hold[2] , nonetheless, concurrently executing operations at different replicas may cause an invariant violation, which we call a *conflict*.

First, we check whether concurrent operations may result in opposite postconditions (e.g., $p(x)$ and $\neg p(x)$), breaking the generic (implicit) invariant that a predicate cannot have two different values. For instance, consider operations $addPlayer(P)$ with effect $player(P)$, vs. $removePlayer(P)$ with effect $\neg player(P)$. These operations conflict, since executing them concurrently with the same parameter P leaves unclear whether player P exists or not in the database. The developer may address this convergence violation by using a conflict resolution policy such as *add-wins* or *remove-wins*.

The remainder of the analysis consists in checking the effect of executing pairs of operations concurrently on the invariant. Our approach is based on Hoare logic [18], where the triple $\{I \wedge P\}\ op\ \{I\}$ expresses that the execution of operation op, in a state where precondition P holds, preserves invariant I. To determine if a set of operations are safe, we substitute their effects on the invariant, obtaining I', and check that the formula I' is valid given that the preconditions to execute the operations hold.

Considering all pairs of operations is sufficient to detect all invariant violations. The intuition why this is correct is that the static analysis considers all possible initial states before executing each concurrent pair, and therefore adding a third concurrent operation is equivalent to modifying the initial state of the two other operations.

To illustrate this process, we consider our tournament application, with the following invariant I:

$$\begin{aligned} I = \forall P, T, &\ enrolled(P,T) \Rightarrow player(P) \wedge tournament(T) \\ &\wedge \\ &nrPlayers(T) \leq 5 \end{aligned}$$

For simplicity of presentation, let us examine each of the conjuncts defined in invariant I separately. First, we consider the numeric restriction: $\forall T, nrPlayers(T) \leq 5$, to illustrate how to check if multiple instances of the same operation are self-conflicting. In this case, one of the operations we need to take into account is $enrollTournament(P,T)$ whose outcome affects $nrPlayers(T)$. This operation has precondition $nrPlayers(T) \leq 4$, the weakest precondition that ensures the sequential execution does not break the invariant (see Footnote 2). To determine if this may break the invariant, we substitute the effects of running the $enrollTournament$ operation twice into invariant I. We then check whether this results in a valid formula, when considering also the weakest precondition. In this example, this corresponds to the following derivation (where notation $I\langle f\rangle$ describes the application of effect f in invariant I):

$$\begin{aligned} I\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\ nrPlayers(T) \leq 5\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\ nrPlayers(T)+1 \leq 5\ \langle nrPlayers(T) \leftarrow nrPlayers(T)+1\rangle \\ nrPlayers(T)+1+1 \leq 5 \end{aligned}$$

[2] This can be achieved by having a precondition such that an operation produces no side effects, if its sequential execution against a state that does not meet that precondition would violate invariants.

Algorithm 1 Algorithm for detecting unsafe operations.

```
Require: $I$ : invariant; $O$ : operations.
 1: $C \leftarrow \emptyset$ {subsets of unsafe operations}
 2: for $op \in O$ do
 3:     if $self\text{-}conflicting(I, \{op\})$ then
 4:         $C \leftarrow C \cup \{\{op\}\}$
 5: for $op, op' \in O$ do
 6:     if $opposing(I, \{op, op'\})$ then
 7:         $C \leftarrow C \cup \{\{op, op'\}\}$
 8: for $op, op' \in O$ : $\{op, op'\} \notin C$ do
 9:     if $conflict(I, \{op, op'\})$ then
10:         $C \leftarrow C \cup \{op, op'\}\}$
11: return $C$
```

The resulting assertion $I' = nrPlayers(T) + 1 + 1 \leq 5$ is not ensured when both the initial invariant and the weakest precondition $nrPlayers(T) \leq 4$ hold. This shows that concurrent executions of $enrollTournament(P, T)$ conflict and $enrollTournament$ is a self-conflicting operation.

The second clause of I is $\forall P, T, enrolled(P, T) \Rightarrow player(P) \wedge tournament(T)$. This case illustrates a conflict between different operations. In this case, we check whether concurrent $enrollTournament(P, T)$ and $removePlayer(P)$ may violate the invariant. Again, we substitute the effects of these operations into the invariant and check whether the resulting formula is valid, assuming that initially the invariant and the preconditions of the two operations hold.

$$I \ \langle enrolled(P,T) \leftarrow true\rangle \ \langle player(P) \leftarrow false\rangle$$
$$enrolled(P,T) \Rightarrow player(P) \wedge tournament(T) \ \langle enrolled(P,T) \leftarrow true\rangle \ \langle player(P) \leftarrow false\rangle$$
$$true \Rightarrow player(P) \wedge tournament(T) \ \langle player(P) \leftarrow false\rangle$$
$$true \Rightarrow false$$
$$false$$

As the resulting formula is not valid, another pair of I-offenders is identified: $\{enrollTournament, removePlayer\}$.

We now present the complete logic to detect I-offender sets in Algorithm 1. This algorithm statically determines the pairs of operation that are conflicting, which are defined as follows.

Definition 4.1 (Conflicting operations)**.** Operations op_1, op_2, $\cdots$, op_n *conflict* with respect to invariant I if, assuming that I is initially true and the preconditions for op_1 and op_2 to produce side effects are initially true, the result of substituting the postconditions of both operations into the invariant is not a valid formula.

The core of the algorithm is made of auxiliary functions, which use the satisfiability modulo theory (SMT) solver Z3 [11] to verify the validity of the logical formulas used in Definition 4.1. Function $self\text{-}conflicting(I, \{op\})$ determines whether op is self-conflicting, i.e., if concurrent executions of op with the same or different arguments may break the invariant. Function $opposing(I, \{op, op'\})$ determines whether op and op' have opposing postconditions. Function $conflict(I, \{op, op'\})$ determines whether the pair of operations break invariant I, by making it false under concurrent execution. They use the solver to check the validity of a set of formulas, namely the invariant, the preconditions for producing effects, and the updated invariant after substituting the effects of both operations.

Algorithm 1 uses these functions for computing I-offender sets in three steps. The initial step (line 2) determines self-conflicting operations. The second step (line 5) determines opposing operations by detecting contradictory predicate assignments for any pair of operations. The last step (line 8) determines other I-offender sets by checking if combining the effects of any two distinct operations raises an invariant violation. If it leads to a conflict, it adds the pair to the set of I-offender sets.

The number of test cases generated is polynomial in the number of operations, $\mathcal{O}(|O|^2)$. However, the satisfiability problem to be solved in each auxiliary function is, in the general case, NP-complete [19]. Z3 relies on heuristics to analyze formulas efficiently, in most cases. The results presented in Section 7.1.1 suggest that it is fast enough to be practical.

5. Handling I-offender sets

The previous step identifies I-offender sets. These sets are reported to the programmer, who decides how each situation should be addressed. We now discuss the techniques that are available to the programmer in Indigo.

5.1 Invariant Repair

One approach is to allow the conflicting operations to execute concurrently, and to repair invariant violations after the fact. Indigo has only limited support for this approach, since it can only address invariants defined in the context of a single database object (even though the object can be complex, such as a tree or a graph). To this end, Indigo provides a library of objects that repair invariants automatically using techniques proposed in the literature, e.g., sets, maps, graphs, trees with different conflict resolution policies [30, 38].

Application developers may extend this library, in order to support additional invariants. For instance, the programmer might want to extend the unbounded set provided by the library, to implement a set with bounded capacity n. She could modify queries such that they ignore excess elements from the underlying unbounded set; however, she must take care to use a deterministic and monotonic algorithm to select the elements to ignore [31].

5.2 Invariant-Violation Avoidance

The alternative approach is to avoid the concurrent execution of operations that would lead to an invariant violation when combining their effects. Indigo provides a set of basic techniques for achieving this, which extend previous ideas from the literature [17, 32, 35, 39, 44]. In comparison to the previous work, we not only combine these ideas in the same system, but we also propose a new implementation, which is optimized for a geo-replicated setting by requiring only peer-to-peer communication, and relying on CRDTs to manage information [38].

5.2.1 Reservations

We now discuss the high-level semantics of the techniques used to restrict the concurrent execution of updates. The next section discusses their implementation in weakly consistent stores.

UID generator: A very common invariant is uniqueness of identifiers [5, 25]. This problem can be easily solved, without coordination, by statically splitting the space of identifiers per replica. Indigo provides this service by appending a replica-specific suffix to a locally-unique identifier.

Multi-level lock reservation: The multi-level lock reservation (or simply multi-level lock) is our base mechanism to restrict the concurrent execution of operations that can break invariants. A multi-level lock can provide the following rights: *(i) shared forbid*, giving the shared right to forbid some action to occur; *(ii) shared allow*, giving the shared right to allow some action to occur; *(iii) exclusive allow*, giving the exclusive right to execute some action.

When a replica holds one of the above rights, no other replica holds rights of a different type. For instance, if a replica holds a *shared forbid*, no other replica has any form of *allow*. We now show how to use this knowledge to control the execution of I-offender sets.

In the tournament example, $\{enrollTournament(P, T),$ $removePlayer(P)\}$ is an I-offender set. To avoid the violation of invariants, we can associate an appropriate multi-level lock to each of the operations, for specific values of the parameters. For example, we can have a multi-level lock associated with $removePlayer(P)$, for each value of P. For executing $removePlayer(P)$, it is necessary to obtain the right *shared allow* on the reservation for $removePlayer(P)$. For executing $enrollTournament(P, T)$, it is necessary to obtain the *shared forbid* right on the reservation for $removePlayer(P)$. This guarantees that enrolling some player will not execute concurrently with deleting the same player. However, concurrent enrolls or concurrent removes are allowed. In particular, if all replicas hold the *shared forbid* right on removing players, the most frequent enroll operation can execute in any replica, without coordination with other replicas.

The *exclusive allow* right, in turn, is necessary when an operation is incompatible with itself, i.e., when executing concurrently the same operation may lead to an invariant violation.

Multi-level locks are a form of lock [17] that can be used to restrict the concurrent execution of operations in any I-offender sets. It would be possible to enforce any application invariants using only multi-level locks. However, in some cases it is possible to provide additional concurrency while enforcing invariants, by using the following reservations.

Multi-level mask reservation: For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates false may lead to an invariant violation if all other predicates were originally false. In our analysis, each of these pairs is an I-offender set.

Using simple multi-level locks for every pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of all operations that could make any of the other predicates false. The reason why this is overly pessimistic is that, in this case, for executing an operation that makes some predicate false it suffices to guarantee that some other predicate remains true, which can be done by only forbidding the operations that make it false.

To allow for this, Indigo includes a multi-level mask reservation that can be seen as a vector of multi-level locks. For the invariant $P_1 \vee P_2 \vee \ldots \vee P_n$, a multi-level mask with n entries is created, with entry i used to control operations that may make P_i false.

When a replica obtains a *shared allow* right in one entry, it must obtain a *shared forbid* right in some other entry. For example, an operation that may make P_i false needs to obtain the *shared allow* right on the i^{th} entry and a *shared forbid* right on an entry j for which the predicate is true. At runtime, to find an entry to forbid, it is only necessary to evaluate the current value of the predicate associated with each entry that can be locked.

Escrow reservation: For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing some decrements to execute without coordination [32]. Given an initial value for $x = x_0$, there are initially $x_0 - k$ rights to execute decrements. These rights can be split dynamically among replicas. For executing $x.decrement(n)$, the operation must acquire and consume n rights to decrement x in the replica it is submitted. If not enough rights exist in the replica, the system will try to obtain additional rights from other replicas. If this is not possible, the operation will fail. Executing $x.increment(n)$ creates n rights to decrement n, initially assigned to the replica in which the operation that executes the increment is submitted.

A similar approach is used for invariants of the form $x \leq k$, with increments consuming rights and decrements creating new rights. For invariants of the form $x + y + \ldots + z \geq k$, a single escrow reservation is used, with decrements to any of the involved variables consuming rights and increments creating rights. If a variable x is involved in more than one invariant, several escrow reservations will be affected by a single increment/decrement operation on x.

The variant called *escrow reservation for conditions* checks a count of elements against some condition; for instance, the number of participants in a tournament in the invariant $nrPlayers(T) < k$. In this case, if the same user is enrolled twice concurrently, two rights are consumed, although the number of participants increases by only one. This is conservative, but "leaks" rights. However, if the same user is disenrolled twice concurrently, then the number of users increases by only one; creating two rights might later let the invariant be violated.

Our escrow reservation for conditions addresses this problem using the following approach (considering invariant $c \geq k$). A decrement operation requires rights, just as a normal escrow reservation. However, an increment operation does not create rights immediately, but instead tags the reservation to be reevaluated. One of the replicas, marked as the primary for the reservation, is entrusted with recreating rights. To do so, it evaluates the distance between the current state and the threshold, taking into account the aggregate number of outstanding rights. More precisely, given the current value for $c = c_1$ and the number k_1 of outstanding rights (i.e., rights assigned to a replica and still not used, as known by the primary replica), $c_1 - k - k_1$ rights are created and assigned initially to the primary replica. This can be done either when the reservation is marked for reevaluation, or when new rights are needed.

Partition lock reservation: For some invariants, it is desirable to have the ability to reserve part of a partitionable resource. For example, consider the invariant that forbids two tournaments to overlap in time. Two operations that schedule different tournaments will break the invariant if the time periods overlap. Using a multi-level lock, it would be necessary to obtain an *exclusive allow* for executing any operation to schedule a new tournament.

However, no invariant violation arises if the time periods of concurrent operations do not overlap. To address this case, we provide a partition lock that allows a replica to obtain an *exclusive lock* on an interval of real values.[3] Replicas can obtain locks on multiple intervals, given that no two intervals reserved by different replicas overlap.

In our example, time would be mapped to a real number. To execute the operation that schedules a tournament, a replica would have to obtain a lock on an interval that includes the time from the start to the end of the tournament.

5.2.2 Using Reservations

The analysis from Section 4 outputs I-offender sets and the corresponding invariant violated. A programmer, electing to use the conflict avoidance approach, must select the type of reservation to be used to avoid invariant violations. Figure 1 presents a default mapping between types of invariants and the corresponding reservations. Conservatively, it is always possible to resort to multi-level locks to enforce any invariant, at the expense of admissible concurrency, as discussed earlier.

[3] Partition locks are a simplified version of partitionable objects [44] and slot reservations [35].

Invariant type	Formula (example)	Reservation
Numeric	$x < K$	Escrow(x)
Referential	$p(x) \Rightarrow q(x)$	Multi-level lock
Disjunction	$p_1 \vee \ldots \vee p_n$	Multi-level mask
Overlapping	$t(s_1, e_1) \wedge t(s_2, e_2) \Rightarrow s_1 \geq e_2 \vee e_1 \leq s_2$	Partition lock
Default	—	Multi-level lock

Table 1. Default mapping from invariants to reservations.

When using multi-level locks to prevent the concurrent execution of I-offender sets, it is possible to use different sets of reservations. We call this a reservation system. For example, consider our tournament application with the following two I-offender sets, which follow from the integrity constraint associated with enrollment: $\{enrollTournament(P, T), removePlayer(P)\}$ and $\{enrollTournament(P, T), removeTournament(P)\}$.

Given these I-offender sets, two alternative reservation systems can be used. The first system includes a single multi-level lock associated with $enroll(P, T)$, where this operation would have to obtain a *shared allow* right to execute, while both $removePlayer(P)$ and $removeTournament(T)$ would have to obtain the *shared forbid* right to execute. The second system includes two multi-level locks associated with $removePlayer(P)$ and $removeTournament(T)$, where enroll would have to obtain the *shared forbid* right in both locks to execute.

A simple optimization process is used to decide which reservations to use. As generating all possible combinations of reservation types may take too long, this process starts by generating a small number of systems using the following heuristic algorithm: *(i)* select a random I-offender set; *(ii)* decide the reservation to control the concurrent execution of operations in the set, and associate the reservation with the operation: if a reservation already exists for some of the operations, use the same reservation; otherwise, generate a new reservation from the type previously selected by the user; *(iii)* select the remaining I-offender set, if any, that has the most operations controlled by existing reservations, and repeat the previous step.

For each generated combination of reservations, Indigo computes the expected frequency of reservation operations needed, using as input the expected frequency of operations. The optimization process tries to minimize this expected frequency of reservation operations.

After deciding which reservation system will be used, each operation is extended to acquire the appropriate rights before executing its code, and to release appropriate rights afterwards. For escrow locks, an operation that consumes rights will acquire rights before its execution (and these rights will not be released when the operation ends). Conversely, an operation that creates rights will create these rights after its execution. For multi-level masks, the programmer must provide the code that verifies the values of the predicate associated with each element of the disjunction.

6. Implementation

In this section, we discuss the implementation of Indigo as a middleware running on top of a causally consistent store. We first explain the implementation of reservations and how they are used to enforce Explicit Consistency. We conclude by explaining how Indigo is designed to use an existing geo-replicated store.

6.1 Reservations

Indigo maintains information about reservations as objects stored in the underlying causally consistent storage system. For each type of reservation, a specific object class exists. Each reservation instance maintains information about the rights assigned to each of the replicas; in Indigo, each datacenter is considered a single replica, as explained later.

The escrow lock object maintains the rights currently assigned to each replica, and the following operations modify its state: *escrow_consume* depletes rights assigned to the local replica; *escrow_generate* generates new rights assigned to the local replica; and *escrow_transfer* transfers rights from the local replica to some given replica. For example, for an invariant $x \geq K$, *escrow_consume* must be used by an operation that decrements x and *escrow_generate* by operations that increment x. For the escrow lock for conditions variant, a replica is tagged as the primary. The *escrow_generate* only creates rights in the primary.

When *escrow_consume* and *escrow_transfer* operations execute in a replica, if that replica has insufficient rights, the operation fails and it has no side effects. Otherwise, the state of the replica is updated accordingly and the side effects are asynchronously propagated to the other replicas, using the normal replication mechanisms of the underlying storage system. As operations only deplete rights of the replica where they are submitted, it is guaranteed that every replica has a conservative view of the rights assigned to it: all operations that have consumed rights are known, but operations that transferred new rights from some other replica may still have to be received. Given that the execution of operations is serialized by the replica, this approach guarantees the correctness of the system in the presence of any number of concurrent updates in different replicas and asynchronous replication, as no replica will ever consume more rights than those assigned to it.

The multi-level lock object maintains which right (exclusive allow, shared allow, shared forbid) is assigned to each replica, if any. Rights are obtained for executing operations with some given parameters. For instance, in the tournament example, for removing player P the replica needs a *shared allow* right for player P. Thus, a multi-level lock object manages the rights for the different parameters independently. Each replica can then hold a given right for a specific value of the parameters or a subset of the parameter values. For simplicity, in our description, we assume that a single parameter exists.

The following operations can be submitted to modify the state of the multi-level lock object: *mll_giveRight* gives a right to some other replica; a replica with a shared right can give the same right to some other replica; a replica that is the only one with some right can change the right type and give it to itself or to some other replica; *mll_freeRight* revokes a right assigned to the local replica. As a replica can have been given rights by multiple concurrent *mll_giveRight* operations executed in different replicas, *mll_freeRight* internally encodes which *mll_giveRight* operations are being revoked. This is necessary to guarantee that all replicas converge to the same state.

As with escrow lock objects, each replica has a conservative view of the rights assigned to it, as all operations that revoke the local rights are always executed initially in the local replica. Additionally, assuming causal consistency, if the local replica shows that it is the only replica with some right, that information is correct system-wide. This condition holds despite concurrent operations and the asynchronous propagation of updates, as any *mll_giveRight* executed in some replica is always propagated before a *mll_freeRight* in that replica. Thus, if the local replica shows that no other replica holds any right, that is because no *mll_giveRight* has been executed (without being revoked).

The multi-level mask object is implemented using a vector of multi-level lock objects, with operations specifying which multi-level lock must be modified.

The partition lock object maintains which replica owns each interval. When it is created, a single replica holds the complete interval of values. A single operation modifies the state of the object: *pol_giveRight*, which transfers part of the interval owned by the local replica to some other replica. Using the same reasoning as in the previous cases, it is clear that the local replica always has a conservative view of the intervals it owns.

6.2 Indigo Middleware

We have built a prototype of Indigo on top of a geo-replicated data store with the following properties: *(i)* causal consistency; *(ii)* support for transactions that access a database snapshot and merge concurrent updates using CRDTs [38]; *(iii)* linearizable execution of operations for each object in each datacenter. There are at least two systems that support all these functionalities: SwiftCloud [46] and Walter [41]. Given that SwiftCloud has a more extensive support for CRDTs, which are fundamental for invariant-repair, we decided to build the Indigo prototype on top of SwiftCloud.

Storing reservations Reservation objects are stored in the underlying storage system and they are replicated in all datacenters. Reservation rights are assigned to datacenters individually, which keeps the information small. As discussed in the previous section, the execution of operations in reservation objects at a given datacenter must be linearizable (to guarantee that two concurrent transactions do not consume the same rights).

The execution of an operation in the replica where it is submitted has three phases: i) the reservation rights needed for executing the operation are obtained; if not all rights can be obtained, the operation fails; ii) the operation executes, reading and writing the objects of the database; iii) the used rights are released (except for escrow reservations, where the rights that are consumed are not released); new rights are created in this step. After the local execution, the side effects of the operation in the data and reservation objects are propagated and executed in other replicas asynchronously and atomically.

Note that reservations guarantee that operations that can lead to invariant violation do not execute concurrently, but they do not guarantee that the preconditions for the operation to generate side effects hold. For example, in the tournament, before removing a tournament it is necessary to disenroll all players, thus guaranteeing that no player in enrolled.

Reservations manager The reservations manager is a service that runs in each datacenter and is responsible for exchanging reservations between datacenters, tracking reservations in use by local clients, and providing clients the database snapshot information to access the underlying storage. For correctness, it is necessary to enforce that updates of an operation are atomic and that reads are causally consistent with the current rights at each replica. In Indigo, these properties are guaranteed directly by the underlying storage system.

An example shows why these properties are necessary. In our tournament application, to enroll a player it is necessary to obtain the right that allows the enroll (by forbidding the removal of both the player and the tournament). After the enroll completes, the right is released and can be obtained by an operation that wants to remove the tournament. The problem is that if the state observed by the remove tournament operation did not include the previous enrollment, the application could end up deleting the tournament without disenrolling the students, leading to an invariant violation.

Obtaining reservation rights The first and last phases of operation execution obtain and free the rights needed for operation execution. Indigo provides API functions for obtaining and releasing a list of rights. Indigo tries to obtain the necessary rights locally using ordered locking to avoid deadlocks. If other datacenters need to be contacted for obtaining some reservation rights, this process is executed before starting to obtain rights locally. Unlike the process for obtaining rights in the local datacenter, Indigo tries to obtain the needed rights from remote datacenters in parallel for minimizing latency. This approach is prone to deadlocks; therefore, if some remote right cannot be obtained, we use an exponential backoff approach that frees all rights and tries to obtain them again after an increasing amount of time.

When it is necessary to contact other datacenters to obtain some right, the latency of operation execution can be severely affected. Therefore, reservation rights are obtained proactively using the following strategy. Escrow lock rights are divided among datacenters, with a datacenter asking for additional rights to the datacenter it believes has more rights (based on local information). The primary of an escrow lock for conditions creates new rights by computing the number of missing rights whenever either it runs out of rights or the object is marked for reevaluation. Multi-level lock and multi-level mask rights are pre-allocated to allow executing the most common operations (based on the expected frequency of operations), with shared allow and forbid rights being shared among all datacenters. In the tournament example, *shared forbid* for removing tournaments and players can be owned in all datacenters, allowing the more frequent enroll operation to execute locally. Partition lock rights are initially assigned to a single replica, and transferred when needed.

The reservations manager maintains a cache of reservation objects and allows concurrent operations to use the same shared (allow or forbid) right. While some ongoing operation is using a shared or exclusive right, the right cannot be revoked. The information about ongoing operations is maintained in soft-state. If the machine where the reservations manager runs fails, the ongoing operation will fail when trying to release the obtained rights.

6.3 Fault tolerance

Indigo builds on the fault tolerance of the underlying storage system. In a typical geo-replicated store, data is replicated inside a datacenter using quorums or a state-machine replication algorithm. Thus, the failure of a machine inside a datacenter does not lead to any data loss. This also applies to the machine running the reservations manager: as explained before, ongoing transactions will fail in this case; committed changes to the reservation objects are stored in the underlying storage system.

If a datacenter (fails or) gets partitioned from other datacenters, it is impossible to transfer rights from and to the partitioned datacenter. In each partition, operations that only require rights available in the partition can execute normally. Operations requiring rights not available in the partition will fail. When the partition is repaired (or the datacenter recovers with its state intact), normal operation is resumed.

In the event that a datacenter fails losing its internal state, the rights held by that datacenter are lost. As reservation objects maintain the rights held by all replicas, the procedure to recover the rights lost by the datacenter failure is greatly simplified: it is only necessary to guarantee that recovery is executed only once with a state that reflects all updates received from the failed datacenter.

7. Evaluation

This section presents an evaluation of Indigo. The main question our evaluation tries to answer is how does Explicit Consistency compares against *causal consistency* and *strong consistency* in terms of latency and throughput with different workloads. Additionally, we try to answer the following questions:

- Can the algorithm for detecting I-offender sets be used with realistic applications?
- What is the impact of an increasing the amount of contention in objects and reservations?
- What is the impact of using an increasing number of reservations in each operation?
- What is the behavior when coordination is necessary for obtaining reservations?

7.1 Applications

To evaluate Indigo, we used the following two applications.

Ad counter The ad counter application models the information maintained by a system that manages ad impressions in online applications. This information needs to be geo-replicated for allowing the fast delivery of ads. For maximizing revenue, an ad should be impressed exactly the number of times the advertiser is willing to pay for. This invariant can be easily expressed as $nrImpressions(A_i) \leq K_i$, where K_i is the maximum number of times ad A_i should be impressed and the function $nrImpressions(A_i)$ returns the number of times it has been impressed.

Advertisers will typically require ads to be impressed a minimum number of times in some countries. For instance, ad A should be impressed exactly 10,000 times, with at least 4,000 impressions in the US and another 4,000 impressions in the EU. This example is modeled through the following invariants for specifying the limits on the number of impressions (where $nrImpressionsOther$ counts the sum of the number of impressions in datacenters other than those two with the impressions in excess of $4,000$ in the EU or the US):

$$nrImpressionsEU(A) \leq 4,000$$
$$nrImpressionsUS(A) \leq 4,000$$
$$nrImpressionsOther(A) \leq 2,000$$

We modeled this application by having one counter for each ad and region pair. Invariants were defined with the target limits stored in the database: $nrImpressions(R, A) \leq targetImpressions(R, A)$ A single update operation that increments the ad tally was defined, which increments the function $nrImpressions$. Our analysis shows that two increment operations for the same counter can lead to an invariant violation, but increments on different counters are independent. Invariants can be enforced by relying on escrow lock reservations for each ad.

Our experiments used workloads with a mix of: a read only operation that returns the value of a set of counters selected randomly; an operation that reads and increments a randomly selected counter. Our default workload included only increment operations.

Tournament management This is a version of the application for managing tournaments described in Section 2 (and used throughout the paper as our running example), extended with read operations for browsing tournaments. The operations defined in this application are similar to operations that one would find in other management applications such as courseware management.

As detailed throughout the paper, this application has a rich set of invariants, including uniqueness rules for assigning ids; generic referential integrity rules for enrollments; and numeric invariants for specifying the capacity of each tournament. This leads to a reservation system that uses both escrow lock for conditions and multi-level lock reservation objects. There are three operations that do not require any right to execute: add player, add tournament and disenroll tournament, although the latter accesses the escrow lock object associated with the capacity of the tournament. The other update operations involve acquiring rights before they can execute.

In our experiments we have run a workload with 82% of read operations (a value similar to the TPC-W shopping workload), 4% of update operations requiring no rights for executing, and 14% of update operations requiring rights (8% of the operations are enrollment and disenrolments).

7.1.1 Performance of the Analysis

We implemented in Java the algorithm described in Section 4 for detecting I-offender sets, relying on the satisfiability modulo theory (SMT) solver Z3 [11] for verifying invariants. As discussed in Section 4, our algorithm relies on the efficiency of Z3 to be able to analyze programs in reasonable time.

Our prototype was was able to find the existing I-offender sets in the applications we have implemented. The average running time of this process in a recent MacBook Pro laptop was 19 ms for the ad counter applications and 730 ms for the more complex tournament application.

For the evaluation of the analysis, we additionally modeled TPC-W, so that we get results for a standard benchmark application. This application has less invariants to check than our custom applications, but has more operations. The running time for detecting I-offender sets was in this case 320 ms. These results show that although the running time increases with the number of invariants and operations, our algorithm can process realistic applications in reasonable times.

7.2 Experimental Setup

We compare Indigo against three alternative approaches:

Causal Consistency (Causal) As our system was built on top of the causally consistent SwiftCloud system [46], we have used unmodified SwiftCloud as representative of a system providing causal consistency. We note that this system cannot enforce invariants. This comparison allows us to measure the overhead introduced by Indigo.

Strong Consistency (Strong) We have emulated a strongly consistent system by running Indigo in a single DC and forwarding all operations to that DC. We note that this approach allows more concurrency than a typical strong consistency system as it allows updates on the same objects to proceed concurrently and be merged if they do not violate invariants.

RedBlue consistency (RedBlue) We have emulated a system with RedBlue consistency [25] by running Indigo in all DCs and having red operations (those that may violate invariants and require reservations) execute in a master DC, while blue operations execute in the closest DC, while respecting causal dependencies.

Our experiments comprised 3 Amazon EC2 datacenters, US-East, US-West and EU, with inter-datacenter latency presented in Table 2. In each DC, Indigo servers run in a single m3.xlarge virtual machine with 4 vCPUs and 8 ECUs of computational power, and 15GB of memory available. Clients that issue transactions run in up to three m3.xlarge machines. Where appropriate, we placed the master DC in the US-East datacenter to minimize the overall communication latency and this way optimize the performance of that configuration.

RTT (ms)	US-E	US-W
US-West	81	–
EU	93	161

Table 2. RTT Latency among datacenters in Amazon EC2

7.3 Latency and Throughput

We start by comparing the latency and throughput of Indigo with alternative deployments for both applications.

We ran the ad counter application with 1000 ads and a single invariant for each ad. The maximum number of impressions was set sufficiently high to guarantee that the limit is not reached. The workload included only update operations for incrementing the counter. This allowed us to measure the peak throughput when operations were able to obtain reservations in advance. The results are presented in Figure 2, and show that Indigo achieves throughput and latency similar to a causally consistent system. Strong and RedBlue results are similar to each other, as all update operations are red and execute in the master DC in both configurations.

Figure 3 presents the results when running the tournament application with the default workload. As before, results show that Indigo achieves throughput and latency similar to a causally consistent system. In this case, as most operations are either read-only or otherwise can be classified as blue and thus execute in the local datacenter, the throughput of RedBlue is only slightly worse than that of Indigo.

Figure 4 details these results, presenting the latency per operation type (for selected operations) in a run with throughput close to the peak value. The results show that Indigo exhibits lower latency than RedBlue for red operations. These operations can execute in the local DC in Indigo, as they require either no reservation or reservations that can be shared and are typically locally available.

Two other results deserve some discussion: *Remove tournament* requires canceling shared forbid rights acquired by other DCs before being able to acquire the shared allow right for removing the tournament, which explain the high latency. Sometimes latency is very high (as shown by the line with the maximum value). This is a result of the asynchronous algorithms implemented and the approach for requesting remote DCs to cancel their rights, which can fail when a right is being used.

Add player has a surprisingly high latency in all configurations. Analyzing the situation, we found out that the reason for this lies in the fact that this operation manipulates very large objects used to maintain indexes, causing all configurations to have a fixed overhead.

7.4 Micro-benchmarks

Next, we examine the impact of key parameters.

Increasing contention Figure 5(a) shows the throughput of the system with increasing contention in the ad counter application, by varying the number of counters in the experiment. As expected, the throughput of Indigo decreases when contention increases as several steps require executing operations sequentially. Furthermore, the results reflect the fact that our middleware introduces an additional level of contention, because operations have to contact the reservation manager.

Increasing number of invariants Figure 5(b) presents the results of the ad counter application with an increasing number of invariants involved in each operation: the operation reads 5 counters (R5) and updates one to three counters (W1 to W3). In this case, the results show that the peak throughput for Indigo decreases while latency keeps constant. The reason for this is that for escrow locks, each invariant has an associated reservation object. Thus, when increasing the number of invariants, the number of updated objects also increases, with an impact on the operations that each datacenter needs to execute. To verify our explanation, we ran a workload with operations that access the same number of counters in the weak consistency configuration. The presented results show the same pattern of decreased throughput.

Impact when transferring reservations Figure 5(c) shows the latency of individual operations executed in the US-W datacenter in the ad counter application, for a workload where increments reach the invariant limit for multiple counters and where the rights were initially assigned to a single

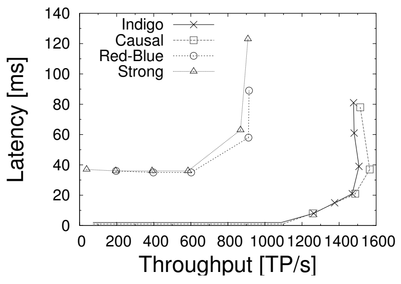


Figure 2. Peak throughput (ad counter application).

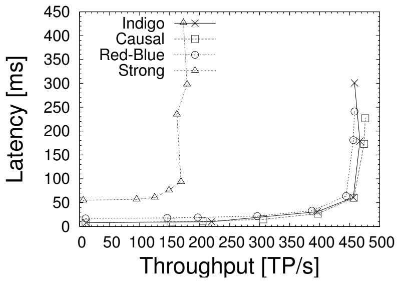


Figure 3. Peak throughput (tournament application).

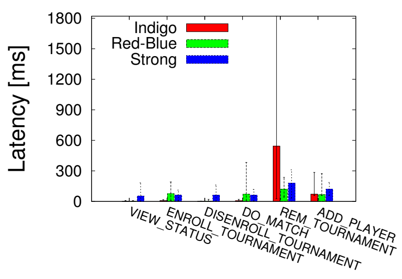
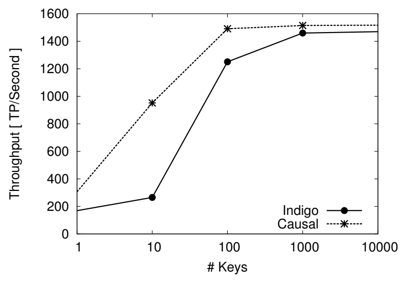
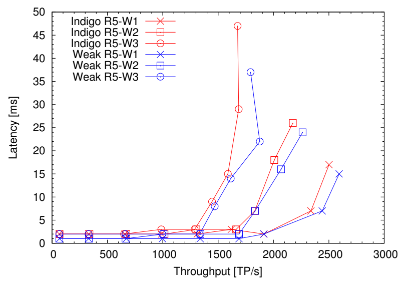


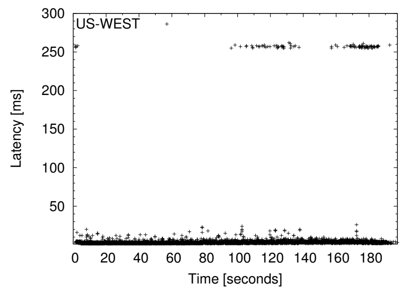
Figure 4. Average latency per op. type - Indigo (tournament app.).



(a) Peak throughput with increasing contention (ad counter application).



(b) Peak throughput with an increasing number of invariants (ad counter application).



(c) Latency of individual operations of US-W datacenter (ad counter application).

Figure 5. Micro-benchmarks.

datacenter. When rights do not exist locally, Indigo cannot mask the latency imposed by coordination, in this case, for obtaining additional rights from the remote datacenters. This explains the high latency operations close to the start of the experiment. As a bulk of rights is obtained, the following operations execute with low latency until it is necessary to obtain additional rights. When a replica believes that no other replica has available rights in an escrow lock object, it does not contact replicas. Instead, the operation fail locally, leading to low latency.

In Figure 4, we showed the impact of obtaining a multi-level lock shared right that requires revoking rights present in all other replicas. We have discussed this problem and a possible solution in Section 7.3. Nevertheless, it is important to note that such impact in latency is only experienced when it is necessary to revoke shared forbid rights in all replicas before acquiring the needed shared allow right. The positive consequence of this approach is that enroll operations requiring the shared forbid right that was shared by all replicas can execute with latency close to zero. The maximum latency line in enroll operation shows the maximum latency experienced when a replica acquires a shared forbid right from a replica already holding such right.

8. Related Work

Geo-replicated storage systems Many cloud storage systems supporting geo-replication emerged in recent years. Some offer variants of eventual consistency, where operations return right after being executed in a single datacenter, usually the closest one, so that end-user response times are improved [2, 12, 23, 27, 28]. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [2, 3, 14, 27]; supporting limited transactions where a set of updates are made visible atomically [4, 28]; supporting application-specific or type-specific reconciliation with no lost updates [7, 12, 27, 41], etc. Indigo is built on top of a geo-replicated store supporting causal consistency, a restricted form of transactions and automatic reconciliation; it extends those properties by enforcing application invariants.

Eventual consistency is insufficient for some applications that require (some operations to execute under) strong consistency for correctness. Spanner provides strong consistency for the whole database, at the cost of incurring coordination overhead for all updates [10]. Transaction chains support transaction serializability with latency proportional to the latency to the first replica that is accessed [47]. MDCC [22] and Replicated Commit [29] propose optimized approaches for executing transactions but still incur in inter-datacenter latency for committing transactions.

Some systems combine the benefits of weak and strong consistency models by supporting both. In Walter [41] and Gemini [25], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [42] and Pileus [43] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [9] and DynamoDB [40] also combine weak consistency with per-object strong consistency relying on conditional writes, where a write fails in the presence of concurrent writes. Indigo enforces Explicit Consistency rules, exploring application semantics to let (most) operations execute in a single datacenter.

Exploring application semantics Several works have explored the semantics of applications (and data types) for improving concurrent execution. Semantic types [16] have been used for building non serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [38] explore commutativity for enabling the automatic merge of concurrent updates, which Walter [41], Gemini [25] and SwiftCloud [46] use as the basis for providing eventual consistency. Indigo goes further by exploring application semantics to enforce application invariants.

Escrow transactions [32] offer a mechanism for enforcing numeric invariants under concurrent execution of transactions. By enforcing local invariants in each transaction, they can guarantee that a global invariant is not broken. This idea can be applied to other data types, and it has been explored for supporting disconnected operation in mobile computing [35, 39, 44]. The demarcation protocol [6] is aimed at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [15] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages.

Indigo builds on these works, but it is the first to provide an approach that, starting from application invariants expressed in first-order logic, leads to the deployment of the appropriate techniques for enforcing such invariants in a geo-replicated weakly consistent data store.

Other related work Bailis et al. [5] studied the possibility of avoiding coordination in database systems and still maintain application invariants. Our work complements that, addressing the cases that cannot entirely avoid coordination, yet allow operations to execute immediately by obtaining the required reservations in bulk and in anticipation.

Others have tried to reduce the need for coordination by bounding the degree of divergence among replicas. Epsilon-serializability [36] and TACT [45] use deterministic algorithms for bounding the amount of divergence observed by an application using different metrics: numerical error, order error and staleness. Consistency rationing [21] uses a statistical model to predict the evolution of replica state and allows applications to switch from weak to strong consistency upon the likelihood of invariant violation. In contrast to these works, Indigo focuses on enforcing invariants efficiently.

The static analysis of code is a standard technique used extensively for various purposes, including in a context similar to ours [8, 13, 20]. Sieve [26] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a RedBlue system [25]. Roy et al. [37] present an analysis algorithm that describes the semantics of transactions. These works are complementary to ours, since the proposed techniques could be used to automatically infer application side effects. The latter work also proposes an algorithm to allow replicas to execute transactions independently by defining conditions that must be met in each replica. Whenever an operation cannot commit locally, a new set of conditions is computed and installed in all replicas using two-phase commit. In Indigo, replicas can exchange rights in a peer-to-peer manner.

9. Conclusions

This paper proposes an application-centric consistency model for geo-replicated services, Explicit Consistency, where programmers specify the consistency rules that the system must maintain as a set of invariants. We describe a methodology that helps programmers decide which invariant-repair and violation-avoidance techniques to use to enforce Explicit Consistency, extending existing applications. We also present the design of Indigo, a middleware that can enforce Explicit Consistency on top of a causally consistent store. The results show that the modified applications have performance similar to weak consistency for most operations, while being able to enforce application invariants. Some rare operations that require intricate rights transfers exhibit high latency. As future work, we intend to improve the algorithms for exchanging reservation rights on those situations.

Acknowledgments

We would like to thank our shepherd Gustavo Alonso, the anonymous reviewers and Alexey Gotsman for their helpful comments on a previous version of this work. This research is supported in part by EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PTDC/ EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014. The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant.

References

[1] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991.

[2] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Com-*

puter Systems, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[3] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. 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] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 27–38, New York, NY, USA, 2014. ACM.

[5] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination Avoidance in Database Systems. *Proc. VLDB Endow. (to appear)*, 2015.

[6] D. Barbará-Millá and H. Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[7] Basho. Riak. `http://basho.com/riak/`, 2014. Accessed Oct/2014.

[8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[11] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. 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.

[13] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 12 1998.

[14] J. Du, S. Elnikety, A. Roy, and W. 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] ed Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for Faster Strong Consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, Berkeley, CA, USA, 2014. USENIX Association.

[16] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[17] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in Database Systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, pages 94–121. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[19] H. B. Hunt and D. J. Rosenkrantz. The Complexity of Testing Predicate Locks. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 127–133, New York, NY, USA, 1979. ACM.

[20] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55. Springer-Verlag, Berlin, Heidelberg, 2011.

[21] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009.

[22] 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*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[23] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2): 35–40, Apr. 2010.

[24] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[25] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[26] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[27] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. 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 Princi-*

ples, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

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

[29] 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.

[30] S. Martin, M. Ahmed-Nacer, and P. Urso. Abstract unordered and ordered trees CRDT. Research Report RR-7825, INRIA, Dec. 2011.

[31] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *Proceedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 471–480. IEEE, Oct 2012.

[32] P. E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[33] S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1975.

[34] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[35] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.

[36] K. Ramamritham and C. Pu. A Formal Characterization of Epsilon Serializability. *IEEE Trans. on Knowl. and Data Eng.*, 7(6):997–1007, Dec. 1995.

[37] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (to appear)*, SIGMOD '15. ACM, May-June 2015.

[38] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[40] S. Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[41] 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*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[42] 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. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[43] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. 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.

[44] G. D. Walborn and P. K. Chrysanthis. Supporting Semantics-based Transaction Processing in Mobile Database Applications. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–40, Washington, DC, USA, 1995. IEEE Computer Society.

[45] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[46] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Research Report RR-8347, INRIA, Oct. 2013.

[47] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

B.3 Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Marc Shapiro, Nuno Preguiça. Extending Eventually Consistent Cloud Stores for Enforcing Numeric Invariants. In Proc. SRDS'15.

Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants

Valter Balegas, Diogo Serra, Sérgio Duarte
Carla Ferreira, Marc Shapiro*, Rodrigo Rodrigues†, Nuno Preguiça
NOVA LINCS, FCT, Universidade NOVA de Lisboa †INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
*Inria Paris-Rocquencourt & Sorbonne Universités, UPMC, LIP6

***Abstract*—Geo-replicated databases often offer high availability and low latency by relying on weak consistency models. The inability to enforce invariants across all replicas remains a key shortcoming that prevents the adoption of such databases in several applications. In this paper we show how to extend an eventually consistent cloud database for enforcing numeric invariants. Our approach builds on ideas from escrow transactions, but our novel design overcomes the limitations of previous works. First, by relying on a new replicated data type, our design has no central authority and uses pairwise asynchronous communication only. Second, by layering our design on top of a fault-tolerant database, our approach exhibits better availability during network partitions and data center faults. The evaluation of our prototype, built on top of Riak, shows much lower latency and better scalability than the traditional approach of using strong consistency to enforce numeric invariants.**

I. Introduction

Scalable cloud databases with a key-value store interface have emerged as the platform of choice for providing online services that operate on a global scale [11], [9], [7]. In this context, a common technique for improving the user experience is geo-replication [9], [7], i.e., maintaining copies of application data and logic in multiple data centers scattered across the globe. This decreases the latency for handling user requests by routing them to a nearby data center, but at the expense of resorting to weaker data consistency guarantees to avoid costly coordination among replicas.

When executing under such weaker consistency models, applications have to deal with concurrent operations. A common approach is to rely on a *last writer wins* strategy [15], [11], but this can lead to lost updates. To address this problem, some databases include specific reconciliation support for some data types, such as counters in Cassandra and DynamoDB, and CRDTs [18] in Riak.

Still, these approaches are unable to enforce invariants across all replicas. For example, it is impossible to enforce numeric invariants (e.g., $x \geq K$), which previous works have shown to be central for maintaining application correctness [14]. This prevents the adoption of such databases in many contexts, such as virtual wallets in games, or management of stocks in e-commerce and ticket reservation applications. In this paper we show how to extend eventually consistent cloud databases for enforcing numeric invariants.

Maintaining this type of invariants would be trivial in systems that offer strong consistency guarantees, namely those that serialize all updates [14], [8]. The problem with these systems is that they require coordination among replicas, leading to an increased latency and reduced fault tolerance. In contrast, our approach builds on the key idea of escrow transactions [16], which is to partition the difference between the current value of a numeric variable and the bound to be enforced among existing replicas. These parts are distributed among replicas, who can locally execute operations that do not exceed their allocated part without contacting other replicas.

In this paper, we present the design of a middleware that overcomes a number of important limitations that exist in previous works that build on the same ideas. First, in contrast to previous escrow based approaches, ours includes no central authority and is totally asynchronous. To this end, we propose a novel replicated data type [18], the *Bounded Counter*, to maintain the information about the escrow each replica holds. Second, we layer the management of *Bounded Counter*s on top of an eventually consistent cloud database. Thus, our design inherits the fault tolerance properties of the underlying database and exhibits better availability than systems that use strong consistency, during network partitions and data center faults. Finally, our middleware combines caching with operation batching, thus improving write throughput without reducing the fault tolerance properties of the system.

The evaluation of our prototype, running on top of Riak, shows that: 1) when compared to using strong consistency, our approach can enforce invariants without paying the latency price for replica coordination, which is considerable for all but the local clients; 2) when compared to using weak consistency, our optimizations lead to higher throughput with a very small increase in latency, while guaranteeing that invariants are not broken.

The remainder of the paper is organized as follows. Section II overviews our approach; Section III introduces the *Bounded Counter* CRDT; Section IV presents our middleware that extends Riak with numeric invariant preservation; Section V evaluates our prototypes; Section VI discusses related work; and Section VII concludes the paper.

II. System Model

We target a typical geo-replicated scenario, with copies of application data and logic replicated in multiple data centers (DCs) scattered across the globe. End clients contact the closest DC for executing application operations. We consider that system processes are connected by an asynchronous network and assume that processes may fail by crashing. A crashed process may either remain crashed forever, or recover with its persistent memory intact.

System API: In addition to *get(key)* and *put(key, value)* operations to access common objects, our middleware provides the following operations to manipulate *Bounded Counter* objects:
(i) *create(key, type, bound)*, creates a new *Bounded Counter* with the given *key*, constraint *type* ($\geq, \leq$) and *bound* – e.g., *create('A', '$\geq$', 10)* creates a counter with initial value 10 that enforces constraint $A \geq 10$;
(ii) *value(key)*, returns the current value of counter *key*;
(iii) *inc(key, value, remote)* and *dec(key, value, remote)*, update the counter if it is known that the change will not break the invariant, with the *remote* flag allowing to request contacting remote nodes if necessary. Update operations return *success* if they succeed or *error* otherwise.

Consistency Guarantees: We build our middleware on top of an eventually consistent database, extending the underlying guarantees with invariant preservation for counters. In particular, the eventual consistency model means that the outcome of each operation reflects the effects of only the subset of operations that have already been executed by the replica that the client has contacted. However, for each operation that successfully returns at a client, there is a point in time after which its effect becomes visible to every operation that is invoked after that time, i.e., operations are eventually executed by all replicas.

In terms of the invariant preservation guarantee, our system guarantees that the value of the counter never violates the bounds specified by the invariant, neither *locally* nor *globally*. By locally, this means that the subset of operations seen by the replica must obey:
lower bound $\leq$ initial value $+\sum inc - \sum dec \leq$ upper bound.
By globally, this means that, at any instant in the execution of the system, when considering the union of all the operations executed by every replica, the same bounds must hold.

Note that the notion of causality is orthogonal to our design, in the sense that if the underlying storage system offers causal consistency, then we also provide numeric invariant-preserving causal consistency.

Enforcing Numeric Invariants: To enforce numeric invariants, our design borrows ideas from the escrow transactional model [16]. The key idea is to see the difference between the value of a counter and its bound as a set of rights to execute operations. Consider, for example, a counter, n, with initial value $n = 40$ and an invariant $n \geq 10$. In this case, there are 30 rights to execute decrement operations. Executing *dec(5)* consumes 5 of these rights. Executing *inc(5)* creates 5 new rights. In this model, these rights can be split among the replicas of the counter. In our example, if there are 3 replicas, each replica can be assigned 10 rights. If the rights needed to execute some operation exist in the local replica, the operation can safely execute locally, knowing that the global invariant will not be broken. Again, in our example, if the decrements of each replica are less or equal to 10, it follows that the total number of decrements will not exceed 30, and therefore the invariant is preserved. If not enough rights exist, then either the operation fails or additional rights must be obtained from other replicas.

Our approach encompasses two components that work together to achieve the goal of our system: a novel data structure, the *Bounded Counter* CRDT, to maintain the necessary information for locally verifying whether it is safe to execute an operation or not (Section III); and a middleware to manipulate instances of this data structure, which are persistently stored in the underlying cloud database (Section IV).

III. Design of Bounded Counter CRDT

This section presents the *Bounded Counter*, a CRDT that maintains information for enforcing numeric invariants without requiring coordination for most executions of operations.

A. CRDT Basics

Conflict-free replicated data types (CRDTs) [18] are a class of distributed data types that allow replicas to be modified without coordination, while guaranteeing that replicas converge to the same value after all updates are propagated and executed in all replicas.

In this work, we adopted the state-based model of CRDTs, as we built our work on top of a key/value store (KV-Store) that synchronizes replicas by propagating the state of the database objects. In this model, an operation submitted in a given site executes in the local replica. Updates are then propagated among replicas in peer-to-peer interactions, where a replica r_1 propagates its state to another replica r_2, which merges its local state with the received state, by executing the *merge()* operation.

It has been proven that a sufficient condition for guaranteeing the convergence of the replicas of state-based CRDTs is that the object conforms the properties of a monotonic semi-lattice object [18], in which: *(i)* The set S of possible states forms a semi-lattice ordered by $\leq$; *(ii)* The result of merging state s with remote state s' is the result of computing the least upper bound (LUB) of the two states in the semi-lattice of states, i.e., $merge(s, s') = s \sqcup s'$; *(iii)* The state is monotonically non-decreasing across updates, i.e., for any update u, $s \leq u(s)$.

B. Bounded Counter CRDT

We now detail the *Bounded Counter*, a CRDT for maintaining the invariant *greater or equal to K*. The pseudocode is presented in Figure 1.

***Bounded Counter* state:** The *Bounded Counter* maintains the limit value K and information about the rights each replica holds. For a system with n replicas, this information is stored in: a matrix R, where entry $R[i][j]$ keeps the rights transferred from replica i to replica j; and in a vector U, where $U[i]$ keeps the rights consumed by replica i.

Operations: An *increment* executed at r_i updates the number of increments for r_i by updating the value of $R[i][i]$. This operation is safe and can always execute locally.

A *decrement* executed at r_i updates the number of decrements for r_i by updating the value of $U[i]$. This operation can only execute if r_i holds enough rights locally before executing the operation, otherwise the operation fails.

The rights of replica r_i, returned by function *localRights*, are given by adding the local increments $R[i][i]$ to the transfers from other replicas to r_i, given by $\sum_{j:j\neq i} R[j][i]$, subtracting the

```
payload integer[n][n] $R$, integer[n] $U$, integer $min$
    initial [[0,0,...,0], ..., [0,0,...,0]], [0,0,...,0], $K$
query $value$ () : integer $v$
    $v = min + \sum_{i \in Ids} R[i][i] - \sum_{i \in Ids} U[i]$
query $localRights$ () : integer $v$
    $id = repId()$                      %Id of the local replica
    $v = R[id][id] + \sum_{i \neq id} R[i][id] - \sum_{i \neq id} R[id][i] - U[id]$
update $increment$ (integer $n$)
    $id = repId()$
    $R[id][id] = R[id][id] + n$
update $decrement$ (integer $n$)
    pre-condition $localRights() \geq n$
    $id = repId()$
    $U[id] = U[id] + n$
update $transfer$ (integer $n$, replicaId $to$): boolean $b$
    pre-condition $b = (localRights() \geq n)$
    $from = repId()$
    $R[from][to] := R[from][to] + n$
update $merge$ (S)
    $R[i][j] = max(R[i][j], S.R[i][j]), \forall i, j \in Ids$
    $U[i] = max(U[i], S.U[i]), \forall i \in Ids$
```

Fig. 1: *Bounded Counter* for invariant *greater or equal to K*.

R	r_1	r_2	r_3	***U***
r_1	30	10	10	5
r_2	0	1	0	4
r_3	0	0	0	2

Limit value (min): 10
Current Value: 20
Local rights:
$r_1 = 5, r_2 = 7, r_3 = 8$

Fig. 2: Example of the state of *Bounded Counter* for maintaining the invariant *greater or equal to 10*.

transfers from r_i to other replicas, $\sum_{j: j \neq i} R[i][j]$, and subtracting the local decrements $U[i]$.

Figure 2 shows an example of a *Bounded Counter* for the invariant *greater or equal to 10*. The initial value of the counter is the bound of the constraint, 10. Replicas r_1, r_2 and r_3 have incremented the counter by 30, 1 and 0 units, respectively, as shown in the diagonal of R. The current value of the counter is given by adding to the limit, the increments performed in every replica, $\sum_i R[i][i]$, and subtracting the decrements, $\sum_i U[i]$, as represented in the grey cells. The operation *transfer* transfers rights from r_i to some other replica r_j, by increasing the value recorded in $R[i][j]$. This operation can only execute if enough local right exist. In the example of Figure 2, transfers of 10 rights from r_1 to each of r_2 and r_3 are recorded in the values of $R[1][2]$ and $R[1][3]$.

The *merge()* operation is executed during peer-to-peer synchronization, when a replica receives the state of a remote replica. The local state is updated by just taking, for each entry, the maximum of the local and the received value.

In a companion technical report [4], we prove that the *Bounded Counter* is a CRDT and that the data structure ensures invariant maintenance in the presence of concurrent updates in different replicas. A TLA proof of correctness is also available.

Extensions: The exact same logic can be applied to preserve invariants of the form *less or equal to K*: Rights represent the possibility of executing *increment* operations instead of *decrement* operations. The specification of the data type is changed accordingly.

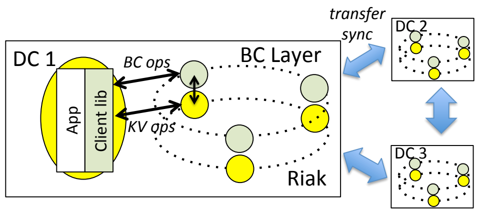


Fig. 3: Middleware for deploying *Bounded Counters*.

Some applications may require two bounds for a counter, e.g., *greater or equal to* K_0 and *less or equal to* K_1. A *Bounded Counter* can maintain an invariant of that form by combining the information of two *Bounded Counter*s in one object, similarly to what is done to specify a PN-Counter from two P-Counters [18]. Some expressions might involve constraints over multiple counters. With the current prototype, the only way to implement these is to store them in a single object, but we do not support it in the current interface.

In general, the approach used for *Bounded Counter*s can be applied to other data types that support escrow [22].

Optimizations: The state of *Bounded Counter* has complexity $O(n^2)$, for n logical replicas. In practice, the impact of this is expected to be small as the number of data centers in common deployments is typically small and each DC will typically hold a single logical replica. In a technical report [4] we show how to lower the space complexity of *Bounded Counters* to $O(n)$.

IV. Middleware to Enforce Numeric Invariants

We now present a middleware, depicted in Figure 3, that uses *Bounded Counter*s to extend cloud databases with numeric invariants. In each DC, our system is composed by a set of middleware nodes and an underlying key-value store to persistently store data. Operations on regular objects execute directly in the key-value store. Operations on counters are handled by middleware nodes, with client requests routed to a specific node using a DHT communication substrate.

In our prototype, we use *riak_core* [10] as the DHT communication substrate and Riak 2.0, a key-value store inspired in Dynamo [9], as the underlying storage system. Riak 2.0 also includes a conditional write mode, where a write from a client fails if there has been a concurrent write since the client's previous operation. Our middleware uses this mechanism to serialize the execution of operations for each counter in each replica. We deploy a logical replica of the *Bounded Counter* per DC, which is replicated in a quorum of nodes by Riak. An operation in a counter is sent to the DHT node responsible for the counter. The DHT node executes the operation by reading the counter from Riak, executing the operation and writing back the new value, using the conditional write mechanism. The operation only succeeds if it is safe, i.e., if the local replica holds enough rights to guarantee the invariant is preserved. By using the conditional write mechanism, we guarantee that operations in each *Bounded Counter* execute sequentially without requiring any guarantees from the DHT. For example, if during a reconfiguration, concurrent requests to the same counter are sent to two different nodes, our approach is still safe as one of the operations will fail when writing to Riak.

Since Riak does not geo-replicate keys marked as strongly consistent, our middleware is responsible for replicating

Bounded Counters across DCs. To this end, each DHT node periodically propagates modified *Bounded Counters* to the remote DCs. When the payload is delivered on the remote DC, it is merged with the local state. This strategy batches a sequence of local operations on a single key and propagates them in a single update, saving bandwidth and processing.

Transferring Rights: Our middleware exchanges rights between replicas in two situations. First, when an operation cannot execute in a replica and the application has specified that remote replicas should be used. In this case, the DHT node executing the operation requests a transfer from a remote DC. To this end, it sends a message to a node in the remote data center, so that it executes a *transfer* operation in the *Bounded Counter*. Second, replicas proactively exchange rights in the background periodically to balance the rights assigned to each replica. These mechanisms are detailed in a separate document [4].

Fault tolerance: We now analyze how our middleware designs provide fault tolerance building on the fault tolerance properties of the underlying cloud database.

The cloud database is assumed to have sufficient internal redundancy to never lose its state in a DC. However, a failure in a node of the middleware layer may cause the DHT to reconfigure, with the possibly that two nodes temporarily accept requests for the same key. This does not affect correctness as we rely on conditional writes to guarantee that operations of each counter are serialized.

During a network partition, rights can be used in both sides of the partition – the only restriction is that it is impossible to transfer rights between any two nodes in different partitions. If an entire DC becomes unavailable, only the rights owned by the unreachable DC become temporarily unavailable. This contrasts with state-of-the-art strong consistency protocols [12], which can only serve requests if at least a majority of replicas (or a primary) is reachable. In our approach, any replica can serve requests if it owns enough rights or if it can gather the needed rights from reachable replicas.

Improving the performance of the middleware: Our prototype includes a number of optimizations to improve its efficiency. The first optimization is to cache *Bounded Counters* on the DHT nodes. This allows us to avoid reading the counter, when it is already in cache. Second, under high contention in a *Bounded Counter*, the design described so far is not very efficient, since an operation must complete before the next operation starts being processed. In particular, since processing an update requires writing the modified *Bounded Counter* back to the Riak database to ensure durability, each operation can take a few milliseconds to complete. To improve throughput, while the write to Riak is taking place, the requests received by the DHT node are processed using the cached counter. The system still writes the batched updates to storage before replying to the waiting clients, but this strategy allows to execute a single write for multiple requests. Our evaluation shows that this strategy improves the throughput of the system by orders of magnitude.

V. Evaluation

We evaluated experimentally our prototype to address the following main questions. (i) How much overhead is introduced by our middleware? (ii) What is the throughput and latency for different levels of contention? (iii) What is the latency when the value is close to the invariant bounds?

A. Configurations and Setup

In the experiments, we compare our middleware, *BC*, with the following configurations:

Weakly Consistent Counters (Weak). This configuration uses Riak 2.0 Enterprise Edition (EE), with native counters running under weak consistency. Native counters handle conflicts automatically inside the database layer. The native geo-replication mechanism of Riak EE is used.

Strongly Consistent Counters (Strong). This configuration runs a Riak 2.0 Community Edition database (for using conditional writes) in a single DC, serving local and remote requests. Updating a counter uses the conditional write mechanism of Riak for enforcing serializability, only succeeding if no concurrent write has completed.

Our experiments comprised 3 Amazon EC2 DCs distributed across the globe. The average latency between DCs is: US-East–Us-West, 80 ms; Europe (EU-West)–US-East, 96 ms; Europe–US-Wast, 160 ms. In each DC, we use three m1.large machines with 7.5GB of memory for running the database servers and server-based middleware and three m1.large machines for running the clients.

Data is fully geo-replicated in all DCs, with clients accessing the replicas in the local DC. Riak operations use a quorum of 3 replicas for writes and 1 replica for reads. In *Strong*, geo-replication is not used, data is stored in the US-East DC, which minimizes the latency for remote clients.

B. Single Counter

We first evaluate performance under high contention. To this end, we use a single counter initialized to a value that is large enough to never break the invariant. Clients execute 20% of increments and 80% of decrements in a closed loop with a think time of 100 ms. Each experiment runs for two minutes after the initialization of the database. The load is controlled by tuning the number of clients running in each experiment, with clients evenly distributed among the client machines.

Throughput vs. latency: Figure 4 presents the variation of the throughput vs. latency values as more operations are injected in the system.

The results of *Strong* show that throughput quickly starts degrading when load increases. This occurs because when more clients try to submit operations to a single DC they increase the interference, which prevents the conditional write from succeeding. We also observe that *Strong* exhibits the higher latency values which occurs because requests are all redirected to a single DC which is remote for $2/3$ of the clients.

In comparison to *Strong*, the throughput of *Weak* is much larger and it does not degrade when increasing the load – after reaching the maximum throughput, increasing the load just leads to an increase in latency. The much higher throughput of the middleware solution is due to the batching mechanism of *BC*, which batches a sequence of updates into a single write to storage. To prove this hypothesis, we ran the same experiment, turning off the batching and writing every update in Riak,

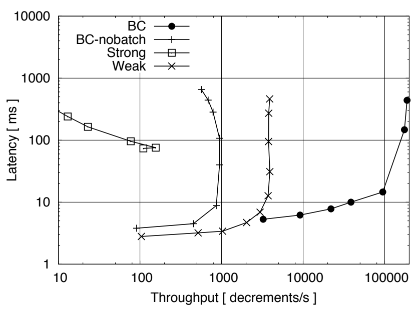


Fig. 4: Throughput vs. latency with a single counter.

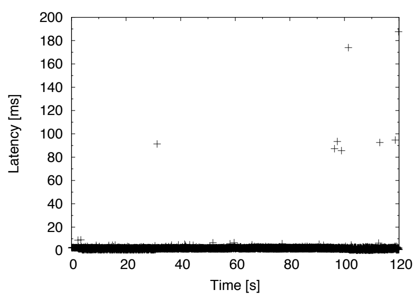


Fig. 5: Latency of each operation over time for BC.

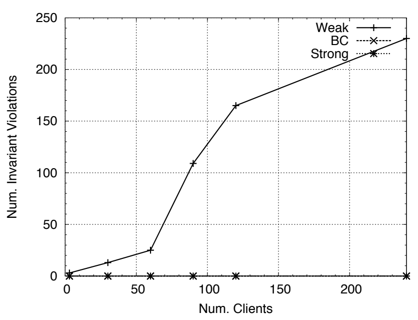


Fig. 6: Decrements executed in excess, violating invariant.

TABLE I: Latency of operations with a single counter.

Median (Max) latency (ms)	Weak	Strong	BC
US-East	2 (7)	172 (180)	4 (9)
US-West	2 (7)	169 (187)	8 (13)
Europe	2 (8)	5 (9)	5 (11)

BC-nobatch. In this case, we can observe that the throughput is much lower than *Weak*, as the middleware introduces an additional communication step and executes operations in sequence. The same approach for batching multiple operations into a single Riak write could be used with other configurations, such as *Weak*, to improve their scalability.

Latency under low load: Table I presents the median and maximum latency experienced by clients in different regions under low load. As expected, the results show that for *Strong*, remote clients experience high latency, while local clients are fast. It also shows that our middleware introduces an overhead of about about 2 ms when compared with *Weak*, which is justified by the additional communication steps.

Effects of exhausting rights: In this experiment we evaluate the behavior of our middleware when the value of the counter approaches the limit and contention for the last available rights rises. We initialize the counter with the value 6000 and 5 clients execute decrement operations until all rights are consumed. Figure 5 shows that most operations have low latency, with a few peaks of high latency whenever a replica needs to obtain additional rights. The number of peaks is small because most of the time the proactive mechanism for exchanging rights is able to provision a replica with enough rights before all local rights are consumed. We see these peaks more frequently near the end of the experiment, because there are less resources available and they might be temporarily exhausted. When all resources are consumed, replicas stop requesting rights and operations fail locally.

Invariant Preservation: To evaluate the severity of the risk of invariant violation, we computed how many decrements in excess were executed with success in the different solutions. We run the same experiment as before, but vary the number of clients. Figure 6 shows that *Weak* is the only configuration that experiences invariant violation. The operation for decrementing consists in reading the counter, checking if the value is greater than the limit and executing a decrement. The decrement operation is not atomic and because of this, multiple decrements can execute concurrently considering the same read value. This effect increases with the number of clients and concurrent updates.

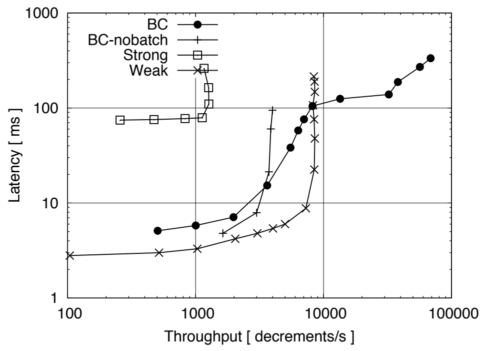


Fig. 7: Throughput vs. latency with multiple counters.

C. Multiple Counters

To evaluate how the system behaves in the common case where clients access to multiple counters, we ran the experiment of Section V-B with 100 counters. For each operation, a client selects the counter to update randomly with uniform distribution. The results presented in Figure 7 show that *Strong* now scales to a larger throughput. The reason for this is that by increasing the number of counters, the number of concurrent writes to the same key is lower, leading to a smaller number of failed operations. Additionally, when the maximum throughput is reached, the latency degrades but the throughput remains almost constant.

The *Weak* configuration scales up to a much larger value (9K decrements/s compared with 3K decrements/s for a single counter). As each Riak node includes multiple virtual nodes, when using multiple counters the load is balanced among them – enabling multi-core capabilities to process multiple requests in parallel (whereas with a single node, a single virtual node is used, resulting in requests being processed sequentially).

The results show that *BC* has a low latency (close to that of *Weak*) as long as the number of writes can be handled by Riak's conditional write mode in a timely manner. In contrast with the experiment with a single counter, Riak's capacity is shared among all the keys, each contributing with writes to Riak. Therefore, as the load increases, writing batches to Riak will take longer to complete and contribute to accumulate latency sooner than in the single key case. Nevertheless, batching still allows multiple client requests to be processed per each Riak operation, leading to a better throughput. The maximum throughput even surpasses the results for the *Weak* configuration.

The results for *BC-nobatch*, where each individual update is written using one Riak operation, can be seen as the worst case of our middleware, in which the batching had no effect. Still, since all *BC* operations are local to a given DC and access only a quorum of Riak nodes, one can expect that increasing the local cluster's capacity should have a positive effect both on latency and throughput.

VI. Related work

Many cloud databases supporting geo-replication have been developed in recent years. Several of them [9], [15], [1], [11], [6], [20] offer variants of eventual/weak consistency where operations return immediately once executed in a single DC. For some applications, strong consistency is necessary to ensure correctness [8]. To avoid the cost of strong consistency for all operations, some systems support both weak and strong consistency for different operation types or objects [14], [7], [20], [6]. In contrast, our work extends eventual consistency with numeric invariants, aiming to keep latency low for all operations.

Bailis et al. [2] examine which invariant of database systems can be enforced without coordination. Indigo [3] extends this approach by providing mechanisms to enforce generic invariants without coordination in most cases. In this work, the focus is on the implementation of a middleware that can be used on top of production databases to provide numeric invariants. We use Riak as a proof of its applicability and show experimentally how to enhance the system's performance by making good use of CRDTs.

Escrow transactions [16], initially proposed for increasing concurrency of transactions in single databases, have also been used for supporting disconnected operation in mobile computing environments either relying on centralized [17], [22] or peer-to-peer [19] protocols for escrow distribution. The demarcation protocol [5] enforces numeric invariants across multiple objects, located in different nodes. Additionally, it shows how to encode other invariants, such as referential integrity, using numeric invariants, which could also be explored in our work. Our work combines convergent data types [18] with ideas from these systems to provide a decentralized approach with replicated data that offers both automatic convergence and invariant preservation with no central authority. Additionally, we describe, implement and evaluate how such solution can be integrated into existing cloud databases.

VII. Conclusion

This paper presents a middleware to extend eventually consistent cloud databases for enforcing numeric invariants. Our design allows most operations to complete within a single DC by moving the necessary coordination outside of the critical path of operation execution. Additionally, our design exhibit a high degree of fault tolerance, by building on the high availability of the underlying database. Thus, we have shown how to combine the benefits of eventual consistency, low latency and high availability, with those of strong consistency, enforcing global numeric invariants. The evaluation of our prototype shows that our middleware has competitive performance when compared with Riak's native weak consistency mechanism where invariants can be compromised.

Acknowledgments

This research is supported in part by EU FP7 SyncFree project (609551), FCT/MCT Phd grant SFRH/BD/87540/2012, SwiftComp project (PTDC/ EEI-SCR/ 1837/ 2012) and NOVA LINCS (UID/CEC/04516/2013). The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant.

References

[1] Almeida, S., Leitão, J. a., and Rodrigues, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. EuroSys '13* (2013).

[2] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination Avoidance in Database Systems. In *Proc. VLDB'15* (2015).

[3] Balegas V., Duarte S., Ferreira C., Preguiça N., Rodrigues R., Najafzadeh M., Shapiro M. Putting Consistency Back into Eventual Consistency. In *Proc. EuroSys '15* (2015).

[4] Balegas, V., Serra, D., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N. M., Shapiro, M., and Najafzadeh, M. Extending eventually consistent cloud databases for enforcing numeric invariants. *CoRR abs/1503.09052* (2015).

[5] Barbará-Millá, D., and Garcia-Molina, H. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal 3* (July 1994).

[6] Basho. Riak. *http://basho.com/riak/*. Accessed Apr/2015.

[7] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1* (Aug. 2008).

[8] Corbett, J. C., Dean et al. Spanner: Google's globally-distributed database. In *Proc. OSDI'12* (2012).

[9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available Key-Value Store. In *Proc. SOSP '07* (2007).

[10] Klophaus, R. Riak core: Building distributed applications without shared state. In *CUFP '10* (2010).

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

[12] Lamport, L. The Part-time Parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[13] Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.

[14] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. OSDI'12* (2012).

[15] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. SOSP '11* (2011).

[16] O'Neil, P. E. The escrow transactional method. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986), 405–430.

[17] Preguiça, N., Martins, J. L., Cunha, M., and Domingos, H. Reservations for conflict avoidance in a mobile database system. In *Proc. MobiSys '03* (2003).

[18] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proc. SSS '11* (2011).

[19] Shrira, L., Tian, H., and Terry, D. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proc. Middleware '08* (2008).

[20] Sivasubramanian, S. Amazon DynamoDB: A seamlessly scalable non-relational database service. In *Proc. SIGMOD '12* (2012).

[21] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proc. SOSP '11* (2011).

[22] Walborn, G. D., and Chrysanthis, P. K. Supporting semantics-based transaction processing in mobile database applications. In *Proc. SRDS '95* (1995).

B.4 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, Marc Shapiro. Towards Fast Invariant Preservation in Geo-replicated Systems. SIGOPS Oper. Syst. Rev. 49, 1 (January 2015), 121-125. ACM.

Towards Fast Invariant Preservation in Geo-replicated Systems

Valter Balegas, Sérgio Duarte
Carla Ferreira, Rodrigo Rodrigues
Nuno Preguiça
NOVA-LINCS
DI/FCT/Univ. Nova de Lisboa

Mahsa Najafzadeh, Marc Shapiro
Inria & UPMC-LIP6

ABSTRACT

Today's global services and applications are expected to be highly available, scale to unprecedented number of clients and offer reliable, low-latency operations. This can be achieved through geo-replication, especially when data consistency is relaxed. There are however applications whose data must obey global invariants at all times. Strong consistency protocols easily address this issue but require global coordination among replicas and inevitably degrade application throughput and latency.

While coordination is an inherent requirement for maintaining global application invariants, there are instances where coordination on a per operation basis can be avoided. In particular, it has been shown that either moving coordination outside the critical path for executing operations, or having one coordination round for multiple operations, are both effective ways to maintain global invariants and avoid most of the penalties of coordination. Our stance is that expanding this idea to geo-replicated settings has yet to be fully realised.

In this paper, we review the design space of current solutions for engineering geo-replicated applications and present our guiding vision towards a general technique for providing global application invariants under eventual consistency, as a much cheaper alternative to strong consistency.

1. INTRODUCTION

The advent of global Internet-based services and applications has fuelled the rise of cloud computing and exposed the challenges of building distributed applications targeting millions of users scattered across the globe. Turning users into customers or potential customers of a whole new economy of social networks and e-commerce platforms made quality of service paramount to achieve success online.

A measure of quality of service that users perceive directly is the responsiveness of their interactions with the service. There is evidence from major industry players [29, 15, 25] that even a slight degradation in latency correlates with increased user dissatisfaction and, consequently, loss of revenue. In recent years, a great deal of research and technology advances have been directed to addressing this issue.

Geo-replication is a widely adopted technique to improve the responsiveness of online services. It employs multiple data centers, placed at strategic locations across the globe, and attempts to redirect user requests to a nearby replica of the service. Thus, the latency between end-users and the servers can be significantly reduced, in addition to offering improvements in system scalability and fault tolerance.

Under geo-replication, systems scale-out by partitioning data requests [14, 5, 12, 18]. Yet, the need to replicate databases over high latency, intercontinental network links forces system designers to choose between system availability and data consistency, since it is not possible to have both under network partitions [10]. Eventually consistent and strongly consistent systems are at the opposite extremes of that trade-off.

Eventually consistent systems forgo tight replica coordination to favor availability, allowing replicas to diverge under network partitions. Operations are executed locally and their effects are replicated asynchronously. This allows users to observe the immediate effects of their actions, but can result in concurrency anomalies, due to conflicting operations performed at other sites. In order to maintain global invariants, applications on top of eventually consistent data stores require additional programming logic, thus complicating their design and development.

Strongly consistent systems, in contrast, are well suited for applications that need to enforce global application invariants across replicas, at all times [9]. In these systems, data consistency is achieved by limiting concurrency, either by funnelling all updates to a central site, or running some consensus algorithm, such as Paxos, so that all sites agree on some global order of operations. However, performing this level of coordination every time the application state is mutated is expensive, particularly in the case of replicas that are far apart, as expected in geo-replication settings. In either case, throughput and scalability are compromised.

In an attempt to bridge the gap between availability and consistency, researchers sought to figure out what guarantees are attainable without impairing availability [3, 34]. They determined that, under some conditions, Causality is the strongest form of always-available consistency [3]. It also happens this is insufficient for enforcing global application invariants, such as ensuring non-negativity of an inventory counter under concurrent decrements.

Others have pursued the approach of combining the best aspects of eventual and strong consistency into systems that choose the most appropriate consistency level for each of the workload operations [33, 22]. Whether that choice is made manually by the programmer (a delicate and error prone process) or by a tool [21, 11, 19], it still remains that the strongly consistent execution path can still undermine availability and performance if those operations are frequent.

While coordination is necessary for enforcing global invariants under concurrency, it should be possible to reap additional parallelism from the following observation: in many cases, operations that in general are unsafe under concurrency, only actually break invariants when particular limit conditions are reached. For instance, when a non-negative counter is far from zero, concurrent decrements do not produce anomalous behaviour, regardless of the order they are committed to the database. In other cases, the typical frequency of unsafe operations in a given application workload may provide an opportunity to save on coordination costs. For instance, the frequency of operations that imperil a referential integrity invariant may be tiny compared to the rest of operations. Treating all these operations in the same way may miss the chance for optimizations - for instance, by requiring the rarer operation to perform most of the burden of global coordination may allow executing the most frequent operation without need to contact other replicas in the common case. These insights have motivated us to improve geo-replication performance in a principled way by moving coordination outside the critical execution path of operations, instead of focusing on the ordering of operations – the approach that is employed by most existing solutions.

The rest of the paper is structured as follows. In section 2, we further discuss the limitations of eventual consistency (EC) using a social network application as an example. Section 3 covers some work on using program analysis to determine which operations require coordination to ensure invariants; then, in Section 4 we review additional techniques for enforcing invariants. Section 5 presents the overall approach of our work for providing global application invariants on top of eventual consistency. Section 6 concludes the paper.

2. PITFALLS OF EVENTUAL CONSISTENCY

Eventual consistency guarantees that *in the future, if updates cease, all replicas will converge to the same value, becoming indistinguishable* [35]. In systems that offer eventual consistency, clients can access any replica, which allows the system to provide high availability despite failures as long as a single replica is available. Additionally, these systems tend to achieve low latency, as the client can access the closest replica. These advantages come at the price of increased complexity in application design [32].

In this section, we use a social network application to illustrate the anomalies that can occur in eventually consistent systems and how to address them by requiring additional guarantees from the system.

In particular, we start by discussing session guarantees [34], which are an interesting set of additional guarantees that can be implemented by eventually consistent systems.

In a social network, a user writes posts that are added to her own wall and to the walls of all her friends. We say that the system provides the *monotonic reads* session guarantee [34] if, after observing some post, successive read operations return a state that includes the post (unless it was explicitly removed). The system provides *read-your-writes* if the client will always reads her previous posts. These guarantees respect only a single user session and can be supported by requiring stick-availability, in which a client maintains *stickness* or *affinity* with a server (or set of servers) [3] or acts as a server by caching the writes and returning them in subsequent reads.

Other session guarantees concern the state observable by any client session. In particular, the system provides the *monotonic writes* guarantee if when a client executes two successive writes, any read that includes the effects of last write also include the effects of the first write.

The final session guarantee is motivated by the fact that, when a post is a reply to a previous post, a user expects to observe the original post before the reply. A system providing the *writes follows reads* guarantee enforces this property – more precisely, if a client does a write w after observing the effects of a set of previous writes S_w, any client that observes the effects of w will also observe the effects of S_w. A system that enforces causality [20] guarantees that all these sessions guarantees are respected, as events are delivered to different replicas according to the happens-before relation. Many recent systems provide causal consistency [23, 1, 24, 38, 5].

In addition to session guarantees, there are other interesting properties that eventually consistent systems may decide to provide. For example, consider the following set of requirements. In social network systems, friendship is usually a bidirectional relation, i.e., if user A is a friend of user B, user B is also friend of user A. As such, when a friend request is acknowledged, both friend lists must be updated. Updating the friend lists without atomicity may result in some user observing that A is friend of B but B is not friend of A or vice-versa. This violates the friendship relation invariant. To address this, some geo-replicated systems provide atomicity for a sequence of writes, while enforcing causality [24, 38, 33].

Finally, as a more challenging requirement, consider the following example scenario. Social networks allow the creation of groups where users can interact. The only invariant that tends to exist is that a user can only join a group for which she has been invited. This rule is easily enforced by using causal consistency, which guarantees that the acceptance of an invitation will always follow the invitation itself. However, stricter semantics would be impossible to enforce relying only on causal consistency, particularly when concurrent operations can lead to a state where the invariant is violated. For example, it is impossible to guarantee that every member of the group is friend of the administrator of the group, since a friendship relationship could be cancelled while a user concurrently joins the group. This invariant can instead be repaired after the violation is detected – e.g., by removing from the group the members that are no longer friends of the administrator.

However, some other invariants may not have a trivial repair function – consider that an award is given to a limited number of users in the group. A system relying on causal consistency could concurrently give out more awards than the limit. In this case, there is no trivial solution to select the users that should remain in the set of awardees, and the situation can be particularly problematic in case the award emails have been sent out.

The examples presented in this section show that there are several additional guarantees that eventually consistent systems should provide. However, in some cases these guarantees can be particularly difficult to enforce under eventual consistency, even with the help of a repair function. As such, to address these requirements, applications tend to adopt strong consistency models (or at least provide support for both weak and strong semantics [33, 22]).

3. MAKING THE RIGHT CHOICE

In the previous section we have seen that not all operations have the same consistency requirements. For this reason, many existing systems take the approach of supporting different levels of consistency to implement applications efficiently.

Gemini [22], Bloom^L [11], Walter [33] and Lazy Replication [19] allow developers to choose between different levels of consistency to ensure application correctness. This approach allows developers to use eventual consistency when operations are compatible with any possible concurrent updates, and only use strong consistency when concurrent operations can make the database inconsistent. This allows for fine tuning the consistency requirements of each operation. However, it poses an heavy burden on the programmer, who must decide the correct level of consistency to use: if the programmer is too conservative, this may lead to an inefficient application; if the programmer is too relaxed due to incorrect reasoning about the application semantics, this can lead to incorrect behaviour. Recent work has proposed to identify the best consistency level automatically, which provides good results free of human error. In particular, Sieve [21] determines the consistency level for operations that run on top of Gemini, under RedBlue consistency. It combines static and dynamic analysis to determine which operations are safe under causal consistency, and which operations need serializability to maintain invariants. The analysis considers a set of user-provided invariants and small annotations that specify the convergence techniques used for concurrent operations on the same objects.

The first step of the analysis, completed offline, generates abstract models that represent the space of possible concurrent executions during runtime and, for each model, determines the set of minimal pre-conditions for being safe to execute the operation without coordination.

At runtime, an operation executes under causal consistency if the minimal pre-conditions for weak execution determined offline are matched. Otherwise, the operation executes under strong consistency.

For example, the offline algorithm would determine that any operation that adds a negative value to a non-negative stock is unsafe to be executed under eventual consistency (as concurrent operations can lead the stock to become negative). At runtime, if an operation adds a positive value, it will execute under eventual consistency; otherwise it needs to execute under strong consistency.

Bloom^L[11] is a logic programming language for distributed applications that maintains application invariants. It is based on the observation that monotonic programs never retract information that is previously known, and therefore they converge regardless the delivery order of messages in different replicas. A total order of messages is only required for non-monotonic operations. An important part of Bloom^L is the CALM analysis that allows to identify which parts of the program are non-monotonic.

The Bloom^L language provides a library of semi-lattice constructs that ensure convergence, similar to CRDTs[30]. The language supports non-monotonic operators: operators that may give different results depending on the arrival order of remote messages. For executing a non-monotonic operator, a coordination protocol must be executed, to ensure that the result of the non-monotonic operation is equivalent in all replicas.

Both strategies identify which operations may break invariants and require coordination among replicas to execute them. This strategy is conservative, as in many executions it is safe to execute the operations without coordination. For instance, in the stock example, coordination is only necessary when the number of available units becomes low, but the system is forced to coordinate on every request because it does not take the current level of the stock into account.

When determining if an operation can execute without coordination. Bloom^L looks only at the code of operations, while Sieve takes into consideration both the code of the operation and the value of parameters. In the latter case, the final decision on whether coordination is necessary or not is executed in runtime. We argue that it is possible to extend this approach by considering also the state of the database. This has the potential to reduce, or even completely avoid, the cost of global coordination by extending conflict analysis with runtime information about the database and the participants.

In the literature, some proposals use the estimation of replica divergence to avoid coordination [37, 17], either by using deterministic or stochastic models. However, these techniques cannot be applied to general invariants and only give a estimation of the divergence, allowing invariants to be broken in certain scenarios.

4. OLD TECHNIQUES REVISITED

In this section, we revisit two works that inspire our vision for enforcing invariants without requiring coordination in the critical path of operation execution: the escrow transactional method [26] and the demarcation protocol [7]. We discuss the use of these protocol to provide the invariants from Section 2 without using strong consistency, or replica coordination in the general case.

The escrow model [26] was proposed to allow long-lived transactions to commit without interfering with other ongoing transactions. The key idea is to divide resources into *escrows* that can be used concurrently by different nodes. If the client has enough resources in its escrow, it can execute the operation without coordination and release the remaining resources on commit, or abort.

In the example of the limited number of awards, consider that each group has a limit of K awards. Each node i that holds a copy of group G grants awards up to a limit Y_i such that $\sum_{i=1}^{n} Y_i <= K$, where n is the number of copies of G. While the number of given awards do not exceed the local limit Y_i, each node can execute the operations locally with low-latency.

This model has been extended to support different partitionable data types [36] and operations [27, 31], but all implementations rely on a central component to manage escrows.

The demarcation protocol [7] has a insight similar to the escrow model, but enforces invariants over multiple variables. For each variable, the protocol defines a limit for the value of the variable. The combination of the defined limits for all variable guarantee that the defined invariants remain valid. Thus, operations are safe if updates do not exceed the defined limits.

If an operations requires a variable to exceeds its limit, another peer must change its limit to make that operation safe: a node sends a request with the change in the limit it requires; the node that accepts the request adjust its own limits and notifies the requester of the change; the requester then increase its safety limits with the received delta and the operation executes safely.

Changing the limits with point-to-point communication can be fast when nodes know enough information about the other peers. When the resources are scarce and nodes change the limits more frequently some request might fail leading to multiple point-to-point messages. Additionally, the point-to-point protocol needs to enforce exactly-once delivery or the limits may become more restrictive than necessary.

The authors have used this protocol to maintain a numeric invariant over resources distributed in multiple machines, enforcing the uniqueness invariant and to provide referential integrity constraints.

A referential integrity constraint is modelled by a logical implication: $predicate(A) \Rightarrow predicate(B)$. Each nodes stores a boolean value for each predicate. The idea is to enforce that whenever a node updates a predicate to a value that may turn the expression false (unsafe), it must enforce that the other nodes changes the value of their predicate to maintain the expression true. In our example, we have $JoinGroup(A, G) \Rightarrow isFriend(A, B)$, with A a user, B the administrator and G a group of users[1]. Making $JoinGroup(A, G)$ *true* is unsafe because that value is only allowed if $isFriend(A, B)$ is true, otherwise the expression is false. The node requests the peer holding the predicate $isFriend(A, B)$ to change the minimum value for that predicate to true. The converse must also be ensured, to make $isFriend(A, B)$ *false* - the node must ask the peer holding the value for predicate $JoinGroup(A, G)$ to ensure it is false.

The idea of distributing data by multiple nodes in an infrastructure has been widely adopted in other contexts to do load balancing for distributed memory multiprocessor [13], quota enforcement in grid [16] and cloud environments [8]. More recently MDCC [18] uses a variation of the demarcation protocol to extract more concurrency of commutative operations that maintain numerical constraints invariants. The homeostasis protocol [28] also extends the demarcation protocol, but requires a new set of conditions to be computed and installed in all replicas using two-phase commit. We argue that it is possible to leverage these old ideas in the new geo-replicated settings relying on peer-to-peer and unreliable asynchronous communication protocols only, as discussed in the next section.

5. LOW-COST INVARIANTS

In the previous sections, we have shown techniques that allow the maintenance of database invariants in two different ways: by identifying what operations are not safe and use strong consistency to execute those or by enforcing local constraints to ensure that operation are safe, while the system is divergent. We argue that a combination of these techniques can be used to provide a principled approach to execute operations that maintain invariants without coordination in the general case.

We envision a system that identifies operations that require strong consistency, but use an efficient protocol to guarantee that local executions are safe instead of using global coordination. The system would exchange the necessary resources, outside the critical path of execution, to guarantee that operations can execute safely, but could still resort to strong consistency when those requirements are not met.

Our preliminary investigations indicates that Sieve is a good candidate to build our system. We could modify the analysis that determines the weakest pre-conditions to accept more facts, computed during runtime, to enable the execution of more operations locally. For instance if the weakest precondition to execute $joingroup(A, G)$ is that $isFriend(A, B)$, than we could add some fact that gives the local replica the exclusive right to modify that predicate, which would ensure it does not become false. On execution, if the current replica holds that guarantee it can execute the operation without coordination because it has the guarantee that the value of that predicate can only change locally. Otherwise, it should resort to strong consistency to execute the operation.

[1]The invariant presented is simplified for illustration purpose, it should also ensure that B is administrator of G

We have a preliminary design of a data-type that maintains numerical invariants [6]. Our data-type maintains the full state of the invariant, which allows the current value to be queried by a client, in opposition to the demarcation protocol, that may require to contact multiple nodes before knowing the actual value of the inequality. Evaluation shows that all operations execute locally while they do not contend for the last available resources. This is a first stepping stone to provide data-types that are able to preserve the demarcation protocol invariants in a replicated system.

Our approach is able to maintain different forms of invariants and we already have a data-type that realises the numerical invariants, but it remains an open question what is the extent of invariants can we capture. Baillis et Al. [4] made a survey on the typical invariants on benchmarks and concluded that the most common invariants have the form of referential integrity, numerical constraints and uniqueness, which all can be implemented with the demarcation protocol.

To our knowledge, none of the previous approaches can be directly applied to implement our vision. None of the former works addresses all the key points in building geo-replicated data-bases: Either they only capture limited forms of invariants, do not deal with data-replication, rely on a central components to manage resources or do not provide low-latency, fault tolerance and scalability to million of clients.

6. CONCLUSION

Current systems give up low-latency and availability for consistency when invariants are essential to applications. At best, only those invariants that are compatible with eventual consistency can be enforced with low latency. For the rest, the default has been to rely blindly on strong consistency. To help figure out which case applies, recent research has produced techniques that help programmers sort out which parts of a program are unsafe under concurrency and need global coordination. Avoiding coordination over expensive inter-continental links has proved to be an important optimisation with noticeable impact on performance.

In this paper, we propose leveraging additional techniques to further avoid paying the full cost of coordination while enforcing global invariants on top of eventual consistency. To the best of our knowledge, no current implementations are tailored to harness these techniques on cloud infrastructures.

After reviewing the literature, we concluded that the approach applies to the most frequent application invariants. We have already confirmed this in part with the design of a data-type that maintains numerical invariants with low-latency. We are now adapting these protocols to be deployed on geo-replicated systems and have been using existing analysis techniques to determine when our optimizations can be applied.

Acknowledgments

This research is supported in part by European FP7 project 609 551 SyncFree (2013–2016), Fundação para a Ciência e Tecnologia SFRH/BD/87540/2012 and PEst-OE/ EEI/ UI0527/ 2014.

7. REFERENCES

[1] S. Almeida, J. Leitão, and L. 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] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination-avoiding database systems. *CoRR*, abs/1402.2237, 2014.

[4] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. 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.

[5] V. Balegas, M. Najafzadeh, S. Duarte, C. Ferreira, rodrod, M. Shapiro, and N. Preguiça. The case for fast and invariant-preserving geo-replication. In *W-PSDS 2014: Workshop on Planetary-Scale Distributed Systems, 2014*. IEEE Computer Society, 10 2014.

[6] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[7] J. Behl, T. Distler, R. Kapitza, and T. Braunschweig. Dqmp: A decentralized protocol to enforce global quotas in cloud environments. In *In Proc. of SSS '12*, pages 217–231, 2012.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[9] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[10] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, New York, NY, USA, 2012. ACM.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[12] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, Oct. 1989.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin,

S. Sivasubramanian, P. Vosshall, and W. 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] P. Dixon. Shopzilla's site redo - you get what you measure. Presented at velocity web performance and operations conference, 2009.

[15] K. Karmon, L. Liss, and A. Schuster. Gwiq-p: An efficient decentralized grid-wide quota enforcement protocol. *SIGOPS Oper. Syst. Rev.*, 42(1):111–118, Jan. 2008.

[16] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009.

[17] 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*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

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

[20] C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. 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.

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

[24] M. Mayer. In search of... a better, faster, stronger web. Presented at velocity web performance and operations conference, 2009.

[25] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[26] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.

[27] S. Roy, L. Kot, N. Foster, J. Gehrke, H. Hojjat, and C. Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014.

[28] E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at velocity web performance and operations conference, 2009.

[29] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[30] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[31] 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. *PVLDB*, 6(11):1068–1079, 2013.

[32] 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*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[33] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. 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.

[34] 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. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[35] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–, Washington, DC, USA, 1995. IEEE Computer Society.

[36] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[37] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR*, abs/1310.3107, 2013.

B.5 Manuel Bravo, Paolo Romano, Luís Rodrigues, and Peter Van Roy. Reducing the Vulnerability Window in Distributed Transaction Protocols. In Proc. PaPoC 2015, 2015. ACM.

Reducing the Vulnerability Window in Distributed Transactional Protocols

Manuel Bravo*†, Paolo Romano†, Luís Rodrigues†, Peter Van Roy*

*Université Catholic de Louvain, Belgium
†INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Abstract

In this paper, we introduce a technique that can be used by distributed transactional protocols to reduce the vulnerability window of transactions. For this purpose, we propose a so far unexplored (to the best of our knowledge) usage of hybrid clocks. On one hand, loosely synchronized physical clocks are used to maximize the freshness of the snapshots used by transactions to read. On the other hand, logical clocks are used to reduce the extent to which the snapshot of update transactions is advanced upon their commit.

We claim that the joint usage of these two techniques can potentially reduce the abort rate in comparison to previous protocols such as Clock-SI, GMU, and SCORe.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed Databases

Keywords hybrid clocks, concurrency control, transactional protocols, abort rate, snapshot isolation

1. Introduction

Capturing the passage of time and the cause-effect relations among events is a key problem at the core of the design of distributed systems. Unsurprisingly, this issue is also of paramount importance in the design of cloud data stores that provide some meaningful consistency guarantee, such as causal consistency [8], snapshot isolation [3], and serializable snapshot isolation [5]. A variety of clock mechanisms have been proposed to track and reason about the order in which events happen, such as physical clocks, logical clocks, and hybrid clocks.

A key characteristic of distributed transactional protocols that impacts the performance of transactional cloud data stores is the abort rate, which is affected by the degree of concurrency. Transaction abort probability depends, naturally, on the workload characteristics. However, the concurrency control mechanism may also play a role in reducing or increasing the likelihood of conflicts. We define the *vulnerability window* as the time window defined between transaction's starting point and its serialization point; other transactions whose *vulnerability window* overlaps may potentially cause the transaction to abort (a more precise definition is given in Section 3). In protocols that use clocks, the *vulnerability window* depends on how the protocol handles time.

In this short paper we propose a novel technique that aims at reducing the *vulnerability window* of transactions. Our technique uses an hybrid clock implementation. The idea is to use the physical part of the hybrid clock to set the starting time of the transaction; therefore, moving the starting point forward in time as much as possible. On the other hand, our technique proposes to use the logical part of the hybrid clock in order to serialize transactions at the earliest possible point in time. The combination of these two techniques has the potential of reducing the *vulnerability window*; and in consequence, the abort rate.

Despite the fact that this is still a work in progress, we believe that this paper already discusses and flags interesting aspects of the use of clocks in distributed transactional protocols. The contributions of this paper are the following:

- A technique that proposes a novel usage of hybrid clocks in distributed transactional protocols that aims at reducing the abort rate by shortening transactions' *vulnerability windows*.
- Comparison and discussion of the implications that different types of clocks pose in the implementation of a distributed transactional protocol. The discussion uses protocols found in the literature such as Clock-SI [4], GMU [12] and SCORe [11].

The rest of the paper is organized as follows. Section 2 gives a brief overview of the different clocks that can be used to order events in distributed systems. Section 3 describes our technique by integrating it into a protocol in order to ease readers comprehension. Section 4 compares our solution to other proposed protocols that use different clock implementations. Finally, Section 5 discusses the next steps of our research and concludes the paper.

2. Clocks

In the design of distributed systems, one could use different clocks techniques to reason about the order of events. A first type of clocks are *physical clocks*. Each participant of a distributed system can use its own physical clock to timestamp events, and reason about the ordering by comparing timestamps. Nevertheless, these clocks can never be perfectly synchronized which may increase system latencies due to the need to keep into account drifts in the clock, e.g., by introducing additional wait phases. Tightly synchronized physical clocks can be achieved by leveraging GPS protocols at the cost of expensive hardware; whereas, loosely synchronized physical clocks can be inexpensively produced by relying on distributed clock synchronization algorithms, such as NTP [10] and PTP [2].

A second type of clocks are *logical clocks*. Introduced by Lamport in 1978 [8], these clocks order events based on passage of in-

PaPoC'15, April 21, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3537-9/15/04…$15.00.
http://dx.doi.org/10.1145/2745947.2746688

formation rather than passage of time. Different forms of logical clocks have been proposed, as scalar [8], vectors [6, 9] and matrix [13, 16]. While scalar clocks are very efficient w.r.t. the message size, they may insert extra dependences between events. Vector and matrix clocks fix this problem at the cost of increasing the size of the messages to sometimes unbearable sizes.

Finally, the last type of clocks are a combination of the previous categories, namely *hybrid clocks*. A good example of this type of clocks is Hybrid Logical Clocks (HLC) [7]. It combines a physical clock with a scalar logical clock. This approach can be used to (i) avoid, at least in some circumstances, waiting periods due to clock drift, and (ii) precisely identify cause-effect relations avoiding the possibility of wrongly ordering events.

3. On Fully Distributed Transactional Protocols

In order to better understand and illustrate the benefits of our technique, we resort to a concrete protocol that embodies it. We have observed that some of the fully distributed transactional protocols in the literature, such as SCORe [11] and Clock-SI [4], share a common structure and mostly only differ for the type of clocks they use. Thus, the protocol we use throughout the discussion shares this common pattern and integrates our technique. In this section, we first give an overview of the protocol and how we integrate our technique. Then, we describe the protocol in detail.

3.1 Protocol Overview

The protocol implements snapshot isolation (SI) [3]. It satisfies the following three properties: (i) each transaction reads from a consistent snapshot, (ii) conflicting update transactions commit in total order producing a new snapshot in the database, and (iii) a transaction aborts if introduces a conflict with a concurrent committed transaction. In SI, two transactions conflict if their write-sets, which is the set of updated data items, have common elements. This type of conflicts are called write-write conflicts. In consequence, SI precludes read-only transactions to abort. Since workloads are usually composed by mostly read-only transactions, SI is likely to improve performance compared to stronger consistency criteria, such as serializability where read-write conflicts abort transactions. SI is the default consistency choice of popular data engines as Oracle and Microsoft SQL Server.

In addition, the protocol can be characterized as a Genuine Partial Replication (GPR) [14] and Deferred Update Replication (DUR) [15] protocol. GPR protocols are those in which only the servers that store data needed by the transaction are involved in the coordination. This is a desirable characteristic for large-scale systems. DUR is an optimization for transactional protocols where updates are buffered in the coordinator and sent atomically in the commit step. This reduces coordination and potentially latency.

The protocol is composed by three phases: (i) an initial phase where transaction's *snapshot time* is set, defining the versions that transactions can read, (ii) an interactive phase where clients issue read and update requests, and (iii) a two phase commit protocol that sets transaction's *commit time*, in case all involved servers agree on committing. We define *vulnerability window* of a transaction as the window time created between transaction's *snapshot time* and transaction's *commit time*. Two transactions whose *vulnerability windows* overlap are considered concurrent by the protocol. Since a transaction is aborted if there is a concurrent conflicting transaction already committed, one goal of this type of protocols should be to shorten the *vulnerability window* as much as possible. This leads to reduce the abort rate and improve protocol's performance.

Our technique precisely focus on this observation. We propose the use of hybrid clocks to identify consistent snapshots and order committed transactions. The hybrid clock is composed by a physical clock and a scalar logical clock. The physical clock is always equal to the value of the server's physical clock and it is used to set transaction's *snapshot time*. We assume that physical clocks of different servers are loosely synchronized through a distributed clock synchronization protocol as NTP; nevertheless, the protocol correctness does not depend on how synchronized clocks are. On the other hand, the scalar logical clock will always be set to the largest time stamp the server has seen. This means that the logical clock is "infected" by the physical time. The protocol uses the logical part of the hybrid clock to propose *commit times*.

3.2 Protocol

Algorithm 1 shows the pseudocode of the protocol running in the coordinator of the transaction (lines 1-24) and on the servers (lines 25-43). Notice that any server can act as a coordinator. A transaction issued by a client would take the following steps:

1. Upon a start transaction request, the coordinator initializes the transaction and sets the snapshot time as the maximum between its physical clock and logical clock (lines 2-5). The *snapshot time* will be used by the transaction to identify the consistent snapshot from where to read.

2. Clients interactively send operations (*read/update*) to the coordinator. Updates are buffered in the coordinator (line 14). Reads are sent to the partition responsible for the data item (if not buffered). Upon a read request for *key*, the server first updates its logical clock (line 26). Then, it waits for prepared conflicting transactions with smaller *prepare time* than transaction's *snapshot time* to commit (lines 27-30). Otherwise, the server may return a version that misses writes of concurrent transactions. Finally, the server returns the largest version with a smaller or equal *commit time* than transaction's *snapshot time*.

3. Upon a commit transaction request, the coordinator starts a two phase commit protocol (2PC) to either *commit* or *abort*.
   - First, the coordinator sends a *prepare* request to the servers storing part of the transactions's write set (lines 17-18).
   - Each server first updates its logical clock (line 33). Then, it waits for already prepared conflicting concurrent transactions to either commit or abort (lines 34-35). Otherwise, SI may be violated. Next, the server runs a certification check that look for conflicting concurrent committed transactions (line 36). If none, the server increases its logical clock (line 37) and uses it as *prepare time*. The proposed *prepare time* is sent to the coordinator. Otherwise, an *abort* message is sent back to the coordinator.
   - The coordinator waits for all the partitions to reply. If all partitions agree on committing, the coordinator sets the *commit time* of the transaction to the maximum of the gathered *prepare times*. Finally, it sends committed to the client and the *commit time* to the involved servers.
   - When a server receives the *commit time*, it applies the updates to its local store using the *commit time* as version id.

Our protocol has two points where the execution may need to be delayed in order to ensure correctness. The first can be found in lines 27-30. A server waits until conflicting concurrent prepared transactions are committed or aborted if their *commit time* may be smaller than current transaction's *snapshot time*. For instance, let us assume two potentially concurrent transactions T_1 and T_2. T_1 starts before T_2, updates data items x and y, and tries to commit in servers P_1 and P_2. On the other hand, T_2 is a read-only transaction that reads data item x in P_1. When the read request reaches P_1, T_1 has not been committed yet; therefore, P_1 does not know whether T_1 has to be included in T_2's snapshot or not. If P_1 proposed a *prepare times* for T_1 smaller than T_2's *snapshot time*, there is a

Algorithm 1: Protocol

```
// Coordinator operations
1  upon receive start_tx() from Client do
2      T.TxId←generate_txid()
3      T.SnapshotTime←max(Server.PhysicalClock, Server.MaxTS)
4      T.State←active
5      T.Client←Client
6      send T to Client

7  upon receive read(T, Key) from Client do
8      if is_buffered(T, Key) then
9          send get_buffered_value(T, Key) to Client
10     else
11         Server←get_responsible(Key)
12         send read(T, Key) to Server

13 upon receive update(T, Key, Value) from Client do
14     buffer_value(T, Key, Value)
15     send ok to Client

16 upon receive commit(T) from Client do
17     foreach Server in T.UpdatePartitions do
18         send prepare(T) to Server
19     wait until receiving PrepareTime from T.UpdatePartitions
20     T.CommitTime←max(all prepare times)
21     T.State←committed
22     foreach Server in T.UpdatePartitions do
23         send commit(T) to Server
24     send ok to Client

// Server operations
25 upon receive read(T, Key) from Coordinator do
26     Server.MaxTS←max(Server.MaxTS, T.SnapshotTime)
27     if Key is updated by T' ∧
28       T'.State = prepared ∧
29       T.SnapshotTime > T'.PrepareTime then
30         wait until T'.State = committed
31     send get(Server.Backend, Key, T.SnapshotTime) to T.Client

32 upon receive prepare(T) from Coordinator do
33     Server.MaxTS←max(Server.MaxTS, T.SnapshotTime)
34     if Key is updated by T' ∧ T'.State = prepared then
35         wait until T'.State = committed
36     if CertificationCheck(T) then
37         Server.MaxTS←Server.MaxTS + 1
38         T.PrepareTime←Server.MaxTS
39         T.State←prepared
40         send T.PrepareTime to Coordinator

41 upon receive commit(T) from Coordinator do
42     T.State←committed
43     put(Server.Backend, T.WriteSet, T.CommitTime)
```

possibility that the maximum of all proposed *prepare time*, and in consequence T_1's *commit time*, is smaller than T_2's *snapshot time*. In this case, T_1 has to be included in T_2's snapshot, otherwise SI is violated. The only way to ensure correctness in this scenario, without adding extra coordination, is to wait for T_1 to finish, as our protocol does. Clock-SI [4], which uses physical clocks to set transactions *snapshot times*, solves the problem similarly.

The second point where waiting can be required is found in lines 34-35. The intuition behind this is that prepared transactions are not considered in the certification check (line 36) and they may pose write-write conflicts, and thus, violate SI. Therefore, we suggest to wait until there is no conflicting transaction committing before starting the certification phase. Let us discuss an example to clarify this safety property. Let us assume two transactions T_1 and T_2 whose write sets intersect in data item x stored in P_1. P_1 receives a prepare request first for T_1. Then, it receives the prepare request for T_2. Since T_1's *commit time* is unknown at this point, there is always the possibility that T_1 and T_2 are concurrent. Therefore, only one should successfully commit. If P_1 do not wait for T_1 to commit or abort before preparing T_2 both may commit, and thus, violate SI. Even when T_1 and T_2 are known to be concurrent, one should not abort T_2 immediately since T_1 may abort.

4. Comparison with Related Work

We now focus on discussing the implications and the trade-offs that our clock choice poses in comparison to other clock mechanisms proposed in the literature. We consider three protocols to compare: SCORe [11] that uses a simple scalar logical clock, GMU [12] that uses a vector clock with an entry per server in the cluster, and Clock-SI [4] that uses a single physical clock. All these protocols share a very similar protocol skeleton to the one described above. In addition, we also use Hybrid Logical Clocks (HLC) [7] in our discussion. In fact, it would be relatively straightforward to use them in our protocol skeleton. Furthermore, HLCs have already been used in transactional databases, such as CockroachDB [1].

As we have seen, there are two crucial points in which the type of clock used characterizes a GPR protocol: assigning the *snapshot time* when the transaction starts and proposing a *commit time* in the commit phase. We analyse them in the following paragraphs.

Assigning snapshot time This step (i) defines how recent the read data is, and (ii) impacts the transaction's *vulnerability window* by setting its starting point. Physical clocks are in general desirable for this task since, with logical clocks, the rate in which each server's clock advances directly depends on how often they participate in transactions. Thus, if a server that was isolated for a while happens to assign the *snapshot time* of a transaction, this is likely to (i) read quite stale data, and (ii) abort since the beginning of the transaction will be set way in the past for active servers. For instance, let us discuss a example with three servers P_1, P_2, and P_3 whose initial logical clocks are the same. After executing a large number of transactions in which only P_1 and P_2 participate, P_3's logical clock will be set way behind in the past in comparison to P_1 and P_2's clocks. In this situation, we say that P_3 is isolated. In consequence, next time that P_3 sets the *snapshot time* of a transaction that updates data items in any of the other servers, the transaction is likely to abort. In the contrary, physical clocks advance automatically even for servers that are isolated by the workload. Thus, physical clocks are capable to avoid both problems. SCORe and GMU suffer from these problems. GMU tackles them by advancing the *snapshot time* as a transaction reads if possible. This, however, comes at the cost of storing and shipping a vector instead of a single scalar.

On the other hand, physical clocks also have a major disadvantage: protocol's performance depends on the clock skew. This has two implications. First, a read request and a prepare request of a transaction with a *snapshot time* in the future (w.r.t. local server's clock) has to be delayed until the local clock catches up. Second, while logical clocks always assign *snapshot times* that represent, at least, already prepared transactions, physical clocks may assign a *snapshot time* that is in the future. This means that a server is more likely to have prepared conflicting transactions that make the snapshot to be unavailable; and thus, delay the transaction (first waiting period of our protocol, lines 27-30). Clock-SI suffers from both problems. On the contrary, our protocol avoids the first by the use of the scalar in conjunction to the physical. Thus, instead of waiting for the physical clock to catch up, our protocol simply updates the logical one. This is possible because *snapshot times* are set as the maximum between the physical and the logical clock. Notice that we are not first to notice this improvement of hybrid clocks over physical clocks, as the HLC paper already mentions it.

Proposing commit time This step impacts the size of transaction's *vulnerability window*. As argued before, the protocol should try to shorten it in order to reduce the abort rate. Thus, there will be less overlapping between the transactions and less chance to find

Protocol	Clocks	Freshness	Vulnerability Window	Unavailable Snapshot	Clock skew
SCORe	Scalar	Low	$f_1(wl)$	No	No
GMU	Vector	Medium	$f_2(wl)$	No	No
Clock-SI	Physical	High	$f_3(wl, cs)$	Yes	Yes
HLC	Hybrid	High	$f_3(wl, cs)$	Yes	No
Our protocol	Hybrid	High	$\leq min(f_1, f_3)$	Yes	No

Table 1. Summary of GPR protocols with different clock choices and its implications. In the *vulnerability window* column, *wl* stands for workload and *cs* stands for clock skew. This column gives an intuition on which factors the size of the *vulnerability window* depends. The last column refers to the technique of delaying transaction's execution to cope with potential clock skews.

conflicts. Based on this assumption, logical clocks are more suitable for this task. They only move forward when necessary while physical clocks automatically advance, potentially proposing larger *commit times*. SCORe and GMU use logical clocks for this task, while Clock-SI uses a physical clock. On the other hand, HLC would take the maximum between the physical clock and the logical clock, potentially leading to similar results than Clock-SI. Our protocol, instead, only uses the logical clock for this task.

Discussion We claim that our protocol takes the best clock choice in both steps, by reducing the *vulnerability window* of transactions and maximizing data freshness. Table 1 summarizes the advantages and disadvantages of different clocks techniques applied to GPR protocols. As the table shows, our protocol is the best among all the protocols. It will (i) serve the most recent data, (ii) generate the smallest *vulnerability windows* (with the exception of GMU that is incomparable), and (iii) avoid points where the execution have to be delayed due to clock skew.

Regarding the size of the *vulnerability window* in other protocols, it will depend on different factors. For instance, in SCORe and GMU, it will depend on how often servers are isolated by the workload. In those scenarios, the *vulnerability window* created for the first transaction after a period of inactivity can be arbitrarily large. Nevertheless, even when the workload does not isolate servers, our protocol will always generate, on average, smaller windows that SCORe since the starting time of the transaction is the maximum between the physical and the logical clock. Thus, if the physical clock is ahead of the logical one, the window's size would be smaller than the one generated by SCORe. On the other hand, if the physical clock is behind, due to clock skew, our protocol will generate window's sizes equivalent to the ones generated by SCORe. Notice that our protocol and GMU are incomparable. Since GMU may advance transaction's *snapshot time*, it may generate smaller windows in some cases.

On the other hand, in Clock-SI and HLC, the size of the *vulnerability window* will depend on the workload and the clock skew. Both would generate the same sizes, since the only improvement of HLC over Clock-SI is that avoids points where the execution has to be delayed due to clock skew. In comparison to our protocol, there are two scenarios to discuss. First, in the hypothetical scenario with perfectly synchronized clocks, our protocol will always generate smaller windows because the logical clock will always be behind the physical one due to network latencies. Thus, the *commit time* of transactions will always be smaller that the ones generated by Clock-SI and HLC. On the other hand, when clocks are only loosely synchronized, if the logical is ahead of the physical one, the three protocols would generate the same window size. Otherwise, our protocol would generate smaller sizes.

5. Future work

We plan to implement the proposed protocol and compare its performance and other parameters, as the abort rate, to other fully distributed transactional protocols. We are mostly interested to compare to systems with a similar protocol but using different type of clocks. This will lead us to experimentally prove or disprove whether our initial conclusions are right.

Acknowledgments

This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n^{o} 609551, the Erasmus Mundus Joint Doctorate Programme under Grant Agreement 2012-0030, and Fundação para a Ciência e Tecnologia (FCT) with reference UID/CEC/50021/2013

References

[1] Cockroach. https://github.com/cockroachdb/cockroach.

[2] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2002*, pages i–144, 2002.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA, 1987.

[4] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *SRDS '13*, pages 173–184, Sept 2013.

[5] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2): 492–528, June 2005.

[6] C. J. Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. 1987.

[7] S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Principles of Distributed Systems*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. 2014.

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

[9] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[10] D. L. Mills. A brief history of ntp time: Memoirs of an internet timekeeper. *ACM SIGCOMM Computer Communication Review*, 33 (2):9–21, 2003.

[11] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 456–475, New York, NY, USA, 2012.

[12] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS '12*, pages 455–465, June 2012.

[13] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *Software Engineering, IEEE Transactions on*, (1):39–47, 1987.

[14] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS '10*, pages 214–224, Washington, DC, USA, 2010.

[15] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012.

[16] G. T. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242. ACM, 1984.

B.6 Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, Victor Fonte. Concise Server-Wide Causality Management for Eventually Consistent Data Stores. In Proc. DAIS 2015, 2015. Springer.

Concise Server-Wide Causality Management for Eventually Consistent Data Stores

Ricardo Gonçalves, Paulo Sérgio Almeida,
Carlos Baquero, and Victor Fonte

HASLab, INESC Tec & Universidade do Minho
Braga, Portugal
{tome,psa,cbm,vff}@di.uminho.pt

Abstract. Large scale distributed data stores rely on optimistic replication to scale and remain highly available in the face of network partitions. Managing data without coordination results in eventually consistent data stores that allow for concurrent data updates. These systems often use anti-entropy mechanisms (like *Merkle Trees*) to detect and repair divergent data versions across nodes. However, in practice hash-based data structures are too expensive for large amounts of data and create too many false conflicts.
Another aspect of eventual consistency is detecting write conflicts. Logical clocks are often used to track data causality, necessary to detect causally concurrent writes on the same key. However, there is a non-negligible metadata overhead per key, which also keeps growing with time, proportional with the node churn rate. Another challenge is deleting keys while respecting causality: while the values can be deleted, per-key metadata cannot be permanently removed without coordination.
We introduce a new causality management framework for eventually consistent data stores, that leverages node logical clocks (Bitmapped Version Vectors) and a new key logical clock (Dotted Causal Container) to provides advantages on multiple fronts: 1) a new efficient and lightweight anti-entropy mechanism; 2) greatly reduced per-key causality metadata size; 3) accurate key deletes without permanent metadata.

Keywords. Distributed Systems, Key-Value Stores, Eventual Consistency, Causality, Logical Clocks, Anti-Entropy.

1 Introduction

Modern distributed data stores often emphasize high availability and low latency [2,9,7] on geo-replicated settings. Since these properties are at odds with strong consistency [3], these systems allow writing concurrently on different nodes, which avoids the need for global coordination to totally order writes, but creates data divergence. To deal with conflicting versions for the same key, generated by concurrent writes, we can either use the *last-writer-wins* rule [5], which only keeps the "last" version (according to a wall-clock timestamp for example) and lose the other versions, or we can properly track each key causal history with logical clocks [10], which track a partial order on all writes for a given key to detect concurrent writes.

Version Vectors [13] – the logical clocks used in Dynamo – are an established technique that provides a compact representation of causal histories [14]. However, Version vectors do not scale well when multiple users concurrently update the same node, as they would require one entry per user. To address this Riak, a commercial Dynamo inspired database, uses a newer mechanism – called *Dotted Version Vectors* [1] – to handle concurrent versions on the same node in addition to the concurrency across nodes. While these developments improved the scalability problem, the logical clock metadata can still be a significant load when tracking updates on lots of small data items.

In this paper, we address the general case in which, for each key, multiple concurrent versions are kept until overwritten by a future version; no updates are arbitrarily dropped. We present a solution that expressively improves the metadata size needed to track per-key causality, while showing how this also benefits anti-entropy mechanisms for node synchronization and add support for accurate distributed deletes[1].

Brief summary of the contributions:

High Savings on Causality Metadata Building on *Concise Version Vectors* [11], and on *Dotted Version Vectors* [1], we present a new causality management framework that uses a new logical clock per node to summarize which key versions are currently locally stored or have been so in the past. With the node clock, we can greatly reduce the storage footprint of keys' metadata by factoring out the information that the node clock already captures. The smaller footprint makes the overall metadata cost closer to last-write-wins solutions and delivers a better metadata-to-payload ratio for keys storing small values, like integers.

Distributed Key Deletion Deleting a key in an eventually consistent system while respecting causality is non-trivial when using traditional version vector based mechanisms. If a key is fully removed while keeping no additional metadata, it will re-appear if some node replica didn't receive the delete (by lost messages or network partitions) and still has an old version (the same applies for backup replicas stored offline). Even worse, if a key is deleted and re-created, it risks being silently overwritten by an older version that had a higher version vector (since a new version vector starts again the counters with zeros). This problem will be avoided by using the node logical clock to create monotonically increasing counters for the key's logical clocks.

Anti-Entropy Eventually consistent data stores rely on anti-entropy mechanisms to repair divergent versions across the key space between nodes. It both

[1] For this work we don't discuss stronger consistency guarantees like client session guarantees or causal consistency across multiple keys, although it is compatible with our framework and it's also part of our future work.

detects concurrent versions and allows newer versions to reach all node replicas. Dynamo [2], Riak [7] and Cassandra [9] use Merkle-trees [12] for their anti-entropy mechanism. This is an expensive mechanism, in both space and time, that requires frequent updates of an hash tree and presents a trade-off between hash tree size and risk of false positives. We will show how a highly compact and efficient node clock implementation, using bitmaps and binary logic, can be leveraged to support anti-entropy and dispense the use of Merkle-trees altogether.

2 Architecture Overview and System Model

Consider a *Dynamo-like* [2] distributed key-value store, organized as large number (e.g., millions) of virtual nodes (or simply nodes) mapped over a set of physical nodes (e.g., hundreds). Each key is replicated over a deterministic subset of nodes – called *replica nodes* for that key –, using for example consistent hashing [6]. Nodes that replicate common keys are said to be *peers*. We assume no affinity between clients and server nodes. Nodes also periodically perform an anti-entropy protocol with each other to synchronize and repair data.

2.1 Client API

At a high level, the system API exposes three operations:

$$\begin{aligned}
\mathsf{read} &: \mathsf{key} && \to \mathcal{P}(\mathsf{value}) \times \mathsf{context},\\
\mathsf{write} &: \mathsf{key} \times \mathsf{context} \times \mathsf{value} && \to (),\\
\mathsf{delete} &: \mathsf{key} \times \mathsf{context} && \to ().
\end{aligned}$$

This API is motivated by the *read-modify-write* pattern used by clients to preserve data causality: the client first reads a key, updates the value(s) and only then writes it back. Since multiple clients can concurrently update the same key, a read operation can return multiple concurrents values for the client to resolve. By passing the read's context back to the subsequent write, every write request provides the context in which the value was updated by the client. This context is used by the system to remove versions of that key already seen by that client. A write to a non-existing key has an empty context. The delete operation behaves exactly like a normal write, but with an empty value.

2.2 Server-side Workflow

The data store uses several protocols between nodes, both when serving client requests, and to periodically perform anti-entropy synchronization.

Serving reads Any node upon receiving a read request can coordinate it, by asking the respective replica nodes for their local key version. When sufficient replies arrive, the coordinator discards obsolete versions and sends to the client the most recent (concurrent) version(s), w.r.t causality. It also sends the causal context for the value(s). Optionally, the coordinator can send the results back to replica nodes, if they have outdated versions (a process known as *Read Repair*).

Serving writes/deletes Only replica nodes for the key being written can coordinate a write request, while non-replica nodes forward the request to a replica node. A coordinator node: (1) generates a new identifier for this write for the logical clock; (2) discards older versions according to the write's context; (3) adds the new value to the local remaining set of concurrent versions; (4) propagates the result to the other replica nodes; (5) waits for configurable number of *acks* before replying to the client. Deletes are exactly the same, but omit step 3, since there is no new value.

Anti-entropy To complement the replication done at write time and to ensure consistency convergence, either because some messages were lost, or some replica node was down for some time, or writes were never sent to all replica nodes to save bandwidth, nodes perform periodically an anti-entropy protocol. The protocol aims to figure out what key versions are missing from which nodes (or must be deleted), propagating them appropriately.

2.3 System Model

All interaction is done via asynchronous message passing: there is no global clock, no bound on the time it takes for a message to arrive, nor bounds on relative processing speeds. Nodes have access to durable storage; nodes can crash but eventually will recover with the content of the durable storage as at the time of the crash. Durable state is written atomically at each state transition. Message sending from a node i to a node j, specified at a state transition of node i by $\mathsf{send}_{i,j}$, is scheduled to happen after the transition, and therefore, after the next state is durably written. Such a send may trigger a $\mathsf{receive}_{i,j}$ action at node j some time in the future. Each node has a globally unique identifier.

2.4 Notation

We use mostly standard notation for sets and maps. A map is a set of (k, v) pairs, where each k is associated with a single v. Given a map m, $m(k)$ returns the value associated with the key k, and $m\{k \mapsto v\}$ updates m, mapping k to v and maintaining everything else equal. The domain and range of a map m is denoted by $\mathsf{dom}(m)$ and $\mathsf{ran}(m)$, respectively. $\mathsf{fst}(t)$ and $\mathsf{snd}(t)$ denote the first and second component of a tuple t, respectively. We use set comprehension of the forms $\{f(x) \mid x \in S\}$ or $\{x \in S \mid Pred(x)\}$. We use $\lhd$ for domain subtraction; $S \lhd M$ is the map obtained by removing from M all pairs (k, v) with $k \in S$. We will use $\mathbb{K}$ for the set of possible keys in the store, $\mathbb{V}$ for the set of values, and $\mathbb{I}$ for the set of node identifiers.

3 Causality Management Framework

Our causality management framework involves two logical clocks: one to be used per node, and one to be used per key in each replica node.

The Node Logical Clock Each node i has a logical clock that represents all locally known writes to keys that node i replicates, including writes to those keys coordinated by other replica nodes, that arrive at node i via replication or anti-entropy mechanisms;

The Key Logical Clock For each key stored by a replica node, there is a corresponding logical clock that represents all current and past versions seen (directly or transitively) by this key at this replica node. In addition, we attached to this key logical clock the current concurrent values and their individual causality information.

While this dual-logical clock framework draws upon the work of Concise Version Vectors (CVV) [11], our scope is on distributed key-value stores (KVS) while CVV targets distributed file-systems (DFS). Their differences pose some challenges which prevent a simple reuse of CVV:

- Contrary to DFS where the only source of concurrency are nodes themselves, KVS have external clients making concurrent requests, implying the generation of concurrent versions for the same key, even when a single node is involved. Thus, the key logical clock in a KVS has to possibly manage multiple concurrent values in a way that preserves causality;
- Contrary to DFS, which considers full replication of a set keys over a set of replicas nodes, in a KVS two peer nodes can be replica nodes for two non-equal set of keys. E.g., we can have a key k_1 with the replica nodes $\{a, b\}$, a key k_2 with $\{b, c\}$ and a key k_3 with $\{c, a\}$; although a, b and c are peers (they are replica nodes for common keys), they don't replicated the exact same set of keys. The result is that, in addition to gaps in the causal history for writes not yet replicated by peers, a node logical clock will have many other gaps for writes to key that this node is not replica node of. This increases the need for a compact representation of a node logical clock.

3.1 The Node Logical Clock

A node logical clock represents a set of known writes to keys that this node is replica node of. Since each write is only coordinated by one node and later replicated to other replica nodes, the n^{th} write coordinated by a node a can be represented by the pair (a, n). Henceforth, we'll refer to this pair as a *dot*. Essentially, a dot is a globally unique identifier for every write in the entire distributed system.

A node logical clock could therefore be a simple set of dots. However, the set would be unbound and grow linearly with writes. A more concise implementation would have a version vector to represent the set of consecutive dots since the first

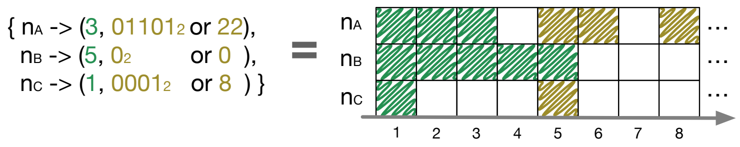


Fig. 1. A bitmapped version vector example and its visual illustration. The bitmap least-significant bit is the first bit from the left.

write for every peer node id, while keeping the rest of the dots as a separate set. For example, the node logical clock: $\{(a, 1), (a, 2), (a, 3), (a, 5), (a, 6), (b, 1), (b, 2)\}$ could be represented by the pair $([(a, 3), (b, 2)], \{(a, 5), (a, 6)\})$, where the first element is a version vector and the second is the set of the remaining dots. Furthermore, we could map peer ids directly to the pair of the maximum contiguous dot and the set of disjointed dots. Taking our example, we have the map: $\{a \mapsto (3, \{5, 6\}), b \mapsto (2, \{\})\}$.

Crucial to an efficient and compact representation of a node logical clock is the need to have the least amount of gaps between dots as possible. For example, the dots in a node logical clock that are from the local node are always consecutive with no gaps, which means that we only need maximum dot counter mapped to the local node id, while the the set of disjointed dots is empty.

The authors of [11] defined the notion of an *extrinsic* set, which we improve and generalize here as follows (note that an event can be seen as a write made to a particular key):

Definition 1 (Extrinsic). *A set of events E_1 is said to be* **extrinsic** *to another set of events E_2, if the subset of E_1 events involving keys that are also involved in events from E_2, is equal to E_2.*

This definition means that we can inflate our node logical clock to make it easier to compact, if the resulting set of dots is extrinsic to the original set. In other words, we can fill the gaps from a node logical clock, if those gaps correspond to dots pertaining to keys that the local node is not replica node of.

Taking this into consideration, our actual implementation of a node logical clock is called *Bitmapped Version Vector* (BVV), where instead of having the disjointed dots represented as a set of integers like before, we use a bitmap where the least-significant bit represents the dot immediately after the dot in the first element of the pair. A 0 means that dot is missing, while a 1 is the opposite. The actual structure of a BVV uses the integer corresponding to the bitmap to efficiently represent large and sparse sets of dots. Figure 3.1 gives a simple BVV example and its visual representation.

Functions over Node Logical Clocks Lets briefly describe the functions necessary for the rest of the paper, involving node logical clocks:

- $\mathsf{norm}(base, bitmap)$ normalizes the pair $(base, bitmap)$. In other words, it removes dots from the disjointed set if they are contiguous to the base, while incrementing the base by the number of dots removed. Example: $\mathsf{norm}(2, 3) = (4, 0)$;
- $\mathsf{values}(base, bitmap)$ returns the counter values for the all the dots represented by the pair $(base, bitmap)$. Example: $\mathsf{values}(2, 2) = \{1, 2, 4\}$;
- $\mathsf{add}((base, bitmap), m)$ adds a dot with a counter m to the pair $(base, bitmap)$. Example: $\mathsf{add}((2, 2), 3) = (4, 0)$;
- $\mathsf{base}(clock)$ returns a new node logical clock with only the contiguous dots from *clock*, i.e., with the bitmaps set to zero. Example: $\mathsf{base}(\{a \mapsto (2, 2), \ldots\}) = \{a \mapsto (2, 0), \ldots\}$;
- $\mathsf{event}(c, i)$ takes the node i's logical clock *clock* and its own node id i, and returns a pair with the new counter for a new write in this node i and the original logical clock c with the new counter added as a dot. Example: $\mathsf{event}(\{a \mapsto (4, 0), \ldots\}, a) = (5, \{a \mapsto (5, 0), \ldots\})$;

Due to size limitations, we refer the reader to appendix A, for a formal definition of a BVV as well as the complete function definitions.

3.2 The Key Logical Clock

A key logical clock using client ids is not realistic in the kind of key-value store under consideration, since the number of clients is virtually unbound. Using simple version vectors with node ids also doesn't accurately capture causality, when a node stores multiple concurrent versions for a single key [1]. One solution is to have a version vector describing the entire causal information (shared amongst concurrent versions), and also associate to each concurrent version their own dot. This way, we can independently reason about each concurrent versions causality, reducing false concurrency. An implementation of this approach can be found in *Dotted Version Vector Sets* (DVVS) [1].

Nevertheless, logical clocks like DVVS are based on per-key information; i.e., each dot generated to tag a write is only unique in the context of the key being written. But with our framework, each dot generated for a write is globally unique in the whole system. One of the main ideas of our framework is to take advantage of having a node logical clock that store these globally unique dots, and use it whenever possible to remove redundant causal information from the key logical clock.

Contrary to version vectors or DVVS, which use per-key counters and thus have contiguous ranges of dots that can have a compact representation, the use of globally unique dots poses some challenges in defining DCC and its operations: even if we only have one version per-key, we still don't necessarily have a contiguous set of dots starting with counter one. Therefore, a compact and accurate implementation of a key logical clock is problematic: using an explicit set of dots is not reasonable as it grows unbounded; neither is using a BVV- like structure, because while a single BVV per node can be afforded, doing so per key is not realistic, as it would result in many low density bitmaps, each as large as the node one. Since there may be millions of keys per node, the size of a key logical clock must be very small.

The solution is to again leverage the notion of extrinsic sets, by filling the gaps in the clock with dots pertaining to other keys, thus not introducing false causal information. The subtlety is that every key logical clock can be inflated to a *contiguous* set of dots, since every gap in the original set was from dots belonging to other keys[2].

Dotted Causal Container Our key logical clock implementation is called Dotted Causal Container (DCC). A DCC is a container-like data structure, in the spirit of a DVVS, which stores both concurrent versions and causality information for a given key, to be used together with the node logical clock (e.g. a BVV). The extrinsic set of dots is represented as a version vector, while concurrents versions are grouped and tagged with their respective dots.

Definition 2. *A* Dotted Causal Container *(*DCC *for short) is a pair* $(\mathbb{I} \times \mathbb{N} \hookrightarrow \mathbb{V}) \times (\mathbb{I} \hookrightarrow \mathbb{N})$*, where the first component is a map from dots (identifier-integer pairs) to values, representing a set of versions, and the second component is a version vector (map from [replica node] identifiers to integers), representing a set extrinsic to the collective causal past of the set of versions in the first component.*

Functions over Key Logical Clocks Figure 2 shows the definitions of functions over key logical clocks (DCC) – which also involves node logical clocks (BVV) – necessary for the rest of the paper. Function values returns the values of the concurrent versions in a DCC; add$(c, (d, v))$ adds all the dots in the DCC (d, v) to the BVV c, using the standard fold higher-order function with the function add defined over BVVs. Function sync merges two DCCs: it discards versions in one DCC made obsolete by the other DCC's causal history, while the version vectors are merged by performing the pointwise maximum. The function context simply returns the version vector of a DCC, which represents the totality of causal history for that DCC (note that the dots of the concurrent versions are also included in the version vector component). Function discard$((d, v), c)$ discards versions in a DCC (d, v) which are made obsolete by a VV c, and also merges c into v. Function add$((d, v), (i, n), x)$ adds to versions d a mapping from the dot (i, n) to the value x, and also advances the i component of the VV v to n.

Finally, functions strip and fill are an essential part of our framework. Function strip$((d, v), c)$ discards all entries from the VV v in a DCC that are covered by the corresponding base component of the BVV c; only entries with greater sequence numbers are kept. The idea is to only store DCCs after stripping the causality information that is already present in the node logical clock. Function fill adds back the dots to a stripped DCC, before performing functions over it.

[2] The gaps are always from other keys, because a node i coordinating a write to a key k that generates a dot (i, n), is guaranteed to have locally coordinated all other versions of k with dots (i, m), where $m < n$, since local writes are handle sequentially and new dots have monotonically increasing counters.

$$
\begin{aligned}
\mathsf{values}((d,v)) &= \{x \mid (\_,x) \in d\}\\
\mathsf{context}((d,v)) &= v\\
\mathsf{add}(c,(d,v)) &= \mathsf{fold}(\mathsf{add}, c, \mathsf{dom}(d))\\
\mathsf{sync}((d_1,v_1),(d_2,v_2)) &= ((d_1 \cap d_2) \cup \{((i,n),x) \in d_1 \cup d_2 \mid n > \min(v_1(i), v_2(i))\},\\
&\qquad \mathsf{join}(v_1,v_2))\\
\mathsf{discard}((d,v),v') &= (\{((i,n),x) \in d \mid n > v'(i)\}, \mathsf{join}(v,v'))\\
\mathsf{add}((d,v),(i,n),x) &= (d\{(i,n) \mapsto x\}, v\{i \mapsto n\})\\
\mathsf{strip}((d,v),c) &= (d, \{(i,n) \in v \mid n > \mathsf{fst}(c(i))\})\\
\mathsf{fill}((d,v),c) &= (d, \{i \mapsto \max(v(i), \mathsf{fst}(c(i))) \mid i \in \mathsf{dom}(c)\})
\end{aligned}
$$

Fig. 2. Functions over Dotted Causal Containers (also involving BVV)

Note that, the BVV base components may have increased between two consecutive strip $\mapsto$ fill manipulation of a given DCC, but those extra (consecutive) dots to be added to the DCC are necessarily from other keys (otherwise the DCC would have been filled and updated earlier). Thus, the filled DCC still represents an extrinsic set to the causal history of the current concurrent versions in the DCC. Also, when nodes exchange keys: single DCCs are filled before being sent; if sending a group of DCCs, they can be sent in the more compact stripped form together with the BVV from the sender (possibly with null bitmaps), and later filled at the destination, before being used. This causality stripping can lead to significant network traffic savings in addition to the storage savings, when transferring large sets of keys.

4 Server-side Distributed Algorithm

We now define the distributed algorithm corresponding to the server-side workflow discussed in section 2.2; we define the node state, how to serve updates (writes and deletes); how to serve reads; and how anti-entropy is performed. It is presented in Algorithm 1, by way of clauses, each pertaining to some state transition due to an action (basically receive), defined by pattern-matching over the message structure; there is also a periodically to specify actions which happen periodically, for the anti-entropy. Due to space concerns, and because it is a side issue, read repairs are not addressed.

4.1 Auxiliary Functions

In addition to the operations over BVVs and DCCs already presented, we make use of: function nodes(k), which returns the replica nodes for the key k; function peers(i), which returns the set of nodes that are peers with node i; function random(s) which returns a random element from set s.

Algorithm 1: Distributed algorithm for node i

```
durable state:
    $g_i$ : BVV, node logical clock; initially $g_i = \{j \mapsto (0,0) \mid j \in \mathsf{peers}(i)\}$
    $m_i : \mathbb{K} \hookrightarrow$ DCC, mapping from a key to its logical clock; initially $m_i = \{\}$
    $l_i : \mathbb{N} \hookrightarrow \mathbb{K}$, log of keys locally updated; initially $l_i = \{\}$
    $v_i$ : VV; other peers' knowledge; initially $v_i = \{j \mapsto 0 \mid j \in \mathsf{peers}(i)\}$
volatile state:
    $r_i : (\mathbb{I} \times \mathbb{K}) \hookrightarrow (\text{DCC} \times \mathbb{N})$, requests map; initially $r_i = \{\}$
on $\mathsf{receive}_{j,i}(\mathsf{write}, k : \mathbb{K}, v : \mathbb{V}, c : \text{VV})$:
    if $i \notin \mathsf{nodes}(k)$ then
        $u = \mathsf{random}(\mathsf{nodes}(k))$                    // pick a random replica node of $k$
        $\mathsf{send}_{i,u}(\mathsf{write}, k, v, c)$                    // forward request to node $u$
    else
        $d = \mathsf{discard}(\mathsf{fill}(m_i(k), g_i), c)$        // discard obsolete versions in $k$'s DCC
        $(n, g_i') = \mathsf{event}(g_i, i)$  // increment and get the new max dot from the local BVV
        $d' = $ if $v \neq \mathsf{nil}$ then $\mathsf{add}(d, (i,n), v)$ else $d$     // if it's a write, add version
        $m_i' = m_i\{k \mapsto \mathsf{strip}(d', g_i')\}$                 // update DCC entry for $k$
        $l_i' = l_i\{n \mapsto k\}$                                  // append key to log
        for $u \in \mathsf{nodes}(k) \setminus \{i\}$ do
            $\mathsf{send}_{i,u}(\mathsf{replicate}, k, d')$       // replicate new DCC to other replica nodes
on $\mathsf{receive}_{j,i}(\mathsf{replicate}, K : \mathbb{K}, d : \text{DCC})$:
    $g_i' = \mathsf{add}(g_i, d)$            // add version dots to node clock $g_i$, ignoring DCC context
    $m_i' = m_i\{k \mapsto \mathsf{strip}(\mathsf{sync}(d, \mathsf{fill}(m_i(k), g_i)), g_i')\}$      // sync with local and strip
on $\mathsf{receive}_{j,i}(\mathsf{read}, K : \mathbb{K}, n : \mathbb{N})$:
    $r_i' = r_i\{(j,k) \mapsto (\{\}, n)\}$                 // initialize the read request metadata
    for $u \in \mathsf{nodes}(k)$ do
        $\mathsf{send}_{i,u}(\mathsf{read\_request}, j, k)$         // request $k$ versions from replica nodes
on $\mathsf{receive}_{j,i}(\mathsf{read\_request}, u : \mathbb{I}, k : \mathbb{K})$:
    $\mathsf{send}_{i,j}(\mathsf{read\_response}, u, k, \mathsf{fill}(m_i(k), g_i))$        // return local versions for $k$
on $\mathsf{receive}_{j,i}(\mathsf{read\_response}, u : \mathbb{I}, k : \mathbb{K}, d : \text{DCC})$:
    if $(u,k) \in \mathsf{dom}(r_i)$ then
        $(d', n) = r_i((u,k))$                               // $d'$ is the current merged DCC
        $d'' = \mathsf{sync}(d, d')$                           // sync received with current DCC
        if $n = 1$ then
            $r_i' = \{(u,k)\} \lhd r_i$              // remove $(u,k)$ entry from requests map
            $\mathsf{send}_{i,u}(k, \mathsf{values}(d''), \mathsf{context}(d''))$                // reply to client $u$
        else
            $r_i' = r_i\{(u,k) \mapsto (d'', n-1)\}$                      // update requests map
periodically:
    $j = \mathsf{random}(\mathsf{peers}(i))$
    $\mathsf{send}_{i,j}(\mathsf{sync\_request}, g_i(j))$
on $\mathsf{receive}_{j,i}(\mathsf{sync\_request}, (n,b) : (\mathbb{N} \times \mathbb{N}))$:
    $e = \mathsf{values}(g_i(i)) \setminus \mathsf{values}((n,b))$         // get the dots from $i$ missing from $j$
    $K = \{l_i(m) \mid m \in e \wedge j \in \mathsf{nodes}(l_i(m))\}$     // remove keys that $j$ isn't replica
    node of
    $s = \{k \mapsto \mathsf{strip}(m_i(k), g_i) \mid k \in K\}$           // get and strip DCCs with local BVV
    $\mathsf{send}_{i,j}(\mathsf{sync\_response}, \mathsf{base}(g_i), s)$
    $v_i' = v_i\{j \mapsto n\}$                               // update $v_i$ with $j$'s information on $i$
    $M = \{m \in \mathsf{dom}(l_i) \mid m < \min(\mathsf{ran}(v_i'))\}$          // get dots $i$ seen by all peers
    $l_i' = M \lhd l_i$                                   // remove those dots from the log
    $m_i' = m_i\{k \mapsto \mathsf{strip}(m_i(k), g_i) \mid m \in M, k \in l_i(m)\}$    // strip the keys removed
    from the log
on $\mathsf{receive}_{j,i}(\mathsf{sync\_response}, g : \text{BVV}, s : \mathbb{K} \hookrightarrow \text{DCC})$:
    $g_i' = g_i\{j \mapsto g(j)\}$                     // update the node logical clock with $j$'s entry
    $m_i' = m_i\{k \mapsto \mathsf{strip}(\mathsf{sync}(\mathsf{fill}(m_i(k), g_i), \mathsf{fill}(d, g)), g_i') \mid (k,d) \in s\}$
```

4.2 Node State

The state of each node has five components: g_i is the node logical clock, a BVV; m_i is the proper data store, mapping keys to their respective logical clocks (DCCs); l_i is a map from dot counters to keys, serving as a log holding which key was locally written, under a given counter; v_i is a version vector to track what other peers have seen of the locally generated dots; we use a version vector and not a BVV, because we only care for the contiguous set of dots seen by peers, to easily prune older segments from l_i corresponding to keys seen by *all* peers; r_i is an auxiliary map to track incoming responses from other nodes when serving a read request, before replying to the client. It is the only component held in volatile state, which can be lost under node failure. All other four components are held in durable state (that must behave as if atomically written at each state transition).

4.3 Updates

We have managed to integrate both writes and deletes in a unified framework. A $\mathsf{delete}(k, c)$ operation is translated client-side to a $\mathsf{write}(k, \mathsf{nil}, c)$ operation, passing a special nil as the value.

When a node i is serving an update, arriving from the client as a $(\mathsf{write}, k, v, c)$ message (first "on" clause in our algorithm), either i is a replica node for key k or it isn't. If it's not, it forwards the request to a random replica node for k. If it is: (1) it discards obsolete versions according to context c; (2) creates a new dot and adds its counter to the node logical clock; (3) if the operation is not a delete ($v \neq \mathsf{nil}$) it creates a new version, which is added to the DCC for k; (4) it stores the new DCC after stripping unnecessary causal information; (5) appends k to the log of keys update locally; (6) sends a replicate message to other replica nodes of k with the new DCC. When receiving a replicate message, the node adds the dots of the concurrent versions in the DCC (but not the version vector) to the node logical clock and synchronizes with local key's DCC. The result is then stripped before storing.

Deletes For notational convenience, doing $m_i(k)$ when k isn't in the map, results in the empty DCC: $(\{\}, \{\})$; also, a map update $m\{k \mapsto (\{\}, \{\})\}$ removes the entry for key k. This describes how a delete ends up removing all content from storage for a given key: (1) when there are no current versions in the DCC; (2) and when the causal context becomes older than the node logical clock, resulting in an empty DCC after stripping. If these conditions are not met at the time the delete was first requested, the key will still maintain relevant causal metadata, but when this delete is known by all peers, the anti-entropy mechanism will remove this key from the key-log l_i, and strip the rest of causal history in the key's DCC, resulting in a complete and automatic removal of the key and all its metadata[3].

[3] The key may not be entirely removed if in the meantime, another client has insert back this key, or made a concurrent update to this key. This is the expected behavior

With traditional logical clocks, nodes either maintained the context of the deleted key stored forever, or they would risk the reappearance of deleted keys or even losing new key-values created after a delete. With our algorithm using node logical clocks, we solve both cases: regarding losing new writes after deletes, updates always have new dots with increasing counters, and therefore cannot be causally in the past of previously deleted updates; in the case of reappearing deletes from anti-entropy with outdated nodes or delayed messages, a node can check if it has already seen that delete's dot in its BVV without storing specific per-key metadata.

4.4 Reads

To serve a read request (third "on" clause), a node requests the corresponding DCC from all replica nodes for that key. To allow flexibility (e.g. requiring a quorum of nodes or a single reply is enough) the client provides an extra argument: the number of replies that the coordinator must wait for. All responses are synchronized, discarding obsolete versions, before replying to the client with the (concurrent) version(s) and the causal context in the DCC. Component r_i of the state maintains, for each pair client-key, a DCC maintaining the synchronization of the versions received thus far, and how many more replies are needed.

4.5 Anti-Entropy

Since node logical clocks already reflect the node's knowledge about current and past versions stored locally, comparing those clocks tells us exactly what updates are missing between two peer nodes. However, only knowing the dots that are missing is not sufficient: we must also know what key a dot refers to. This is the purpose of the l_i component of the state: a log storing the keys of locally coordinated updates, which can be seen as a dynamic array indexed by a contiguous set of counters.

Periodically, a node i starts the synchronization protocol with one of its peers j. It starts by sending j's entry of i's node logical clock to j. Node j receives and compares that entry with its own local entry, to detect which local dots node i hasn't seen. Node j then sends back its own entry in its BVV (we don't care about the bitmap part) and the missing key versions (DCCs) that i is also replica node of. Since we're sending a possibly large set of DCCs, we stripped them of unnecessary causal context before sending, to save bandwidth (they were stripped when they where stored, but the node clock has probably advanced since then, so we strip the context again to possibly have further savings).

Upon reception, node i updates j's entry in its own BVV, to reflect that i has now seen all updates coordinated by j reflected in j's received logical clock. Node i also synchronizes the received DCCs with the local ones: for each key, its

when dealing with concurrent writes or new insertions after deletes. Excluding these concurrent or future writes, eventually all keys that received a delete request will be removed.

	Key/Leaf Ratio	Hit Ratio	Per Node Metadata Exchanged	Metadata Per Key Repaired
Merkle Tree	1	60.214 %	449.65 KB	4.30 KB
	10	9.730 %	293.39 KB	2.84 KB
	100	1.061 %	878.40 KB	7.98 KB
	1000	0.126 %	6440.96 KB	63.15 KB
BVV & DCC	–	100 %	3.04 KB	0.019 KB

	Average Entries Per Key L. Clock
VV or DVV	3
DCC	0.231

Table 1. Results from a micro-benchmark run with 10000 writes.

fills the received DCC with j's logical clock, it reads and fills the equivalent local DCCs with i's own logical clock, and then synchronizes each pair into a single DCC and finally locally stores the result after striping again with i's logical clock.

Additionally, node j also: (1) updates the i's entry in v_j with the max contiguous dot generated by j that i knows of; (2) if new keys are know known by all peers (i.e. if the minimum counter of v_j has increased), then remove the corresponding keys from the key-log l_i. This is also a good moment to revisit the locally saved DCCs for these keys, and check if we can further strip causality information, given the constant information growth in the node logical clock. As with deletes, if there were no new updates to a key after the one represented by the dot in the key-log, the DCC will be stripped of its entire causal history, which means that we only need one dot per concurrent version in the stored DCC.

5 Evaluation

We ran a small benchmark, comparing a prototype data store[4] based on our framework[5], against a traditional one based on Merkle Trees and per-key logical clocks[6]. The system has 16 nodes and was populated with 40000 keys, each key replicated in 3 nodes, and we measured some metrics over the course of 10000 writes, 10% losing a message replicating the write to one replica node. The evaluation aimed to compare metadata size of anti-entropy related messages and the data store causality-related metadata size. We compared against four Merkle Trees sizes to show how its "resolution", i.e., the ratio of keys-per-leaf impacts results.

Table 1 shows the results of our benchmark. There is always significant overhead with Merkle Trees, worse for larger keys-per-leaf ratios, where there are many false positives. Even for smaller ratios, where the "hit ratio" of relevant-hashes over exchanged-hashes is higher, the tree itself is large, resulting in substantial metadata transferred. In general, the metadata overhead to perform anti-entropy with our scheme is orders of magnitude smaller than any of the Merkle Tree configurations.

Concerning causality-related metadata size, being negligible the cost of node-wide metadata amortized over a large database, the average per-key logical clock

[4] `https://github.com/ricardobcl/DottedDB`
[5] `https://github.com/ricardobcl/GlobalLogicalClocks`
[6] `https://github.com/ricardobcl/BasicDB`

metadata overhead is also significantly smaller in our scheme, since most of the time the causality is entirely condensed by the node-wide logical clock. With traditional per-key logical clocks, the number of entries is typically the degree of replication, and can be larger, due to entries for retired nodes that remain in the clock forever, a problem which is also solved by our scheme.

6 Related Work

The paper's mechanisms and architecture extend the specialized causality mechanisms in [11,1], apply it over a eventually consistent data store. In addition to the already mentioned differences between our mechanism and Concise Version Vectors [11], our key logical clock size is actually bounded by the number of active replica nodes, unlike PVEs (the CVV key logical clock is unbounded).

Our work also builds on concepts of weakly consistent replication present in log-based systems [15,8,4] and data/file synchronization [13]. The assignment of local unique identifiers for each update event is already present in [15], but each node totally orders its local events, while we consider concurrent clients to the same node. The detection of when an event is known in all other replicas nodes – a condition for log pruning – is common to the mentioned log-based systems; however, our log structure (the key log) is only an inverted index that tracks divergent data replicas, and thus is closer to optimistic replicated file-systems. Our design can reduce divergence both as a result of foreground user activity (both on writes, deletes, and read repair) and by periodic background anti-entropy, while using a common causality framework.

7 Conclusion

The mechanisms and architecture introduced here significantly reduce the metadata size for eventually consistent data stores. This also makes logical clocks systems more competitive with systems that use last-writer-wins and ignore causality. The lightweight anti-entropy mechanism introduced removes a traditional bottleneck in these designs, that used merkle-tree with heavy maintenance and resulted in false positives overhead for data divergence. Finally, the proposed modeling of deletes and how they deal with (re-)creation of keys, provides a simple solution to a non-trivial problem in distributed systems.

Further work will extend the framework to allow variations that address integration of stronger session guaranties and causal consistency, while keeping a general approach that is still expressive enough to keep concurrent versions and comply with no-lost-updates.

References

1. Almeida, P.S., Baquero, C., Gonçalves, R., Preguiça, N., Fonte, V.: Scalable and accurate causality tracking for eventually consistent stores. In: Distributed Applications and Interoperable Systems (DAIS 2014). pp. 67–81. Springer (2014)

2. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: ACM SIGOPS Operating Systems Review. vol. 41, pp. 205–220 (2007)
3. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51–59 (2002)
4. Golding, R.A.: Weak-consistency group communication and membership. Ph.D. thesis, University of California Santa Cruz (1992)
5. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (1976)
6. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. pp. 654–663. ACM (1997)
7. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming. ACM (2010)
8. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. 10(4), 360–391 (Nov 1992)
9. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
11. Malkhi, D., Terry, D.: Concise version vectors in winfs. In: Distributed Computing, pp. 339–353. Springer (2005)
12. Merkle, R.C.: A certified digital signature. In: Proceedings on Advances in Cryptology. pp. 218–238. CRYPTO '89, Springer-Verlag New York, Inc., New York, NY, USA (1989), http://dl.acm.org/citation.cfm?id=118209.118230
13. Parker Jr, D.S., Popek, G., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D.: Detection of mutual inconsistency in distributed systems. IEEE Transactions on Software Engineering pp. 240–247 (1983)
14. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: In search of the holy grail. Distributed computing 7(3), 149–174 (1994)
15. Wuu, G., Bernstein, A.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC). pp. 233–242 (1984)

A Node Logical Clock Implementation: Bitmapped Version Vector

A Bitmapped Version Vector is a version vector augmented with a bitmap per entry; here we use arbitrary size integers for the bitmap components.

Definition 3. *A* Bitmapped Version Vector *(*BVV *for short) is a map from identifiers to pairs of integers,* $\mathbb{I} \hookrightarrow \mathbb{N} \times \mathbb{N}$*, where an entry* $\{i \mapsto (n, b)\}$ *represents the set of the* m^{th} *events from node* i *such that* $m \leq n$ *or* $b \gg (m-1-n) \bmod 2 \neq 0$.

Here $\gg$ stands for the right-bitshift operator, in a C-like notation. In a BVV, the first integer in each pair plays the same role as the integer in a version vector, i.e., a base component, representing a downward-closed set of events, with no gaps, and the second component is interpreted as a bitmap, describing events with possible gaps, where the least-significant bit represents the event after those given by the base component. As an example, the BVV $\{i \mapsto (4, 10)\}$ represents the set of events from node i given by $\{i_1, i_2, i_3, i_4, i_6, i_8\}$, as $10 = 1010_2$, which means that the event following i_4, i.e., i_5 is missing, as well as i_7.

In a BVV, as gaps after the base are filled, the base moves forward, and thus keeps the bitmap with a reasonable size. The idea is that as time passes, the base will describe most events that have occurred, while the bitmap describes a relatively small set of events. The base describes in fact a set of events that is extrinsic to the events relevant to the node, and its progress relies on the anti-entropy algorithm. In a BVV's bitmap there is no point in keeping set bits representing events contiguous to the base; pairs of integers in BVVs are normalized by a norm function, making the second integer in the pair always an even number.

In Figure 3 we define functions over BVVs[7], where c ranges over BVV clocks, i over identifiers, n and m over natural numbers, and b over natural numbers playing the role of bitmaps; operator or denotes bitwise or.

[7] The presentation aims for clarity rather than efficiency; in actual implementations, some optimizations may be performed, such as normalizing only on word boundaries, e.g., when the first 64 bits of a bitmap are set.

$$
\begin{aligned}
\mathsf{norm}(n, b) &= \begin{cases} \mathsf{norm}(n+1, b \gg 1) & \text{if } b \bmod 2 \neq 0, \\ (n, b) & \text{otherwise.} \end{cases} \\
\mathsf{values}((n, b)) &= \{m \in \mathbb{N} \mid m \leq n \vee (b \gg (m-1-n)) \bmod 2 \neq 0\} \\
\mathsf{add}((n, b), m) &= \begin{cases} \mathsf{norm}(n, b) & \text{if } m \leq n, \\ \mathsf{norm}(n, b \mathbin{\mathsf{or}} (1 \ll (m-1-n))) & \text{otherwise.} \end{cases} \\
\mathsf{add}(c, (i, n)) &= c\{i \mapsto \mathsf{add}(c(i), n)\} \\
\mathsf{base}(c) &= \{(i, (n, 0)) \mid (i, (n, \_)) \in c\} \\
\mathsf{event}(c, i) &= (n, \mathsf{add}(c, (i, n))) \qquad \text{where } n = \mathsf{fst}(c(i)) + 1
\end{aligned}
$$

Fig. 3. Operations over Bitmapped Version Vectors

B.7 Ali Shoker, Paulo Sérgio Almeida, Carlos Baquero. Exactly-Once Quantity Transfer. In Proc. W-PSDS'15.

Exactly-Once Quantity Transfer

Ali Shoker, Paulo Sérgio Almeida and Carlos Baquero
HASLab / INESC TEC & University of Minho
Braga, Portugal

***Abstract*—Strongly consistent systems supporting distributed transactions can be prone to high latency and do not tolerate partitions. The present trend of using weaker forms of consistency, to achieve high availability, poses notable challenges in writing applications due to the lack of linearizability, e.g., to ensure global invariants, or perform mutator operations on a distributed datatype. This paper addresses a specific problem: the exactly-once transfer of a "quantity" from one node to another on an unreliable network (coping with message duplication, loss, or reordering) and without any form of global synchronization. This allows preserving a global property (the sum of quantities remains unchanged) without requiring global linearizability and only through using pairwise interactions between nodes, therefore allowing partitions in the system. We present the novel quantity-transfer algorithm while focusing on a specific use-case: a redistribution protocol to keep the quantities in a set of nodes balanced; in particular, averaging a shared real number across nodes. Since this is a work in progress, we briefly discuss the correctness of the protocol, and we leave potential extensions and empirical evaluations for future work.**

***Keywords*-Distributed monoid-like data-types; exactly-once quantity-transfer, idempotence.**

I. Introduction

The trend of distributed storage systems nowadays is to use relaxed forms of consistency to improve availability. This is often established through delaying inter-replica synchronization and offering the requesting client a fast (though stale) response based on the local state, that is coordinated with other replicas off the critical path in an asynchronous fashion. In order to relax consistency in a way that is tolerated by application semantics, that semantics needs to be considered. In this paper, we focus on *monoid*-like datatypes that hold partitionable quantities, that can be split and added back, such as counters or multi-sets.

In the simplest formulation, we consider the distributed datatype state to depict a *quantity*, say a collection of tickets, that is partitioned among a set of nodes. Contrary to replicated systems where the same *total* value is present at all replicas, here the local quantity is a *part* of the whole, and can be immediately operated upon, e.g., increased or decreased by local requests, with no need for node synchronization, resulting in high availability and low latency. Local operations only depend on the quantity locally available and, by being conservative, a global invariant can be preserved as a result from a local invariant: if a decrease is limited to the local quantity, it will remain non-negative, and therefore, so will the sum of all quantities in the system.

Over time, the quantities can become unbalanced across nodes: excess of tickets on some nodes and scarcity on others. This motivates the asynchronous transfer of quantities between nodes in order to balance them. The transfer can be performed pairwise, opportunistically, without requiring global connectivity, and therefore with part of the system being partitioned. The challenge of this approach is how to perform the transfer reliably, with an exactly-once guarantee, to preserve the total quantity in the system.

Many *redistribution* protocols (e.g., [1], [2], [3], [4], [5], [6]) have been proposed to redistribute quantities, however none was immune to message duplication, i.e., the messages involved were not idempotent. In this work, we propose an new redistribution (a.k.a., quantity transfer) protocol with idempotent messages.

Redistribution protocols in the 80's suffered either from latency issues due to resource locking and extensive use of 2PC (two-phase commit) or from delivery ordering constraints [2], [1], [7]. The demarcation protocol [3], [4] was then proposed as an alternative solution that is immune to message delays and reception order: Whenever a node wishes to perform an unsafe operation (e.g., may violate an invariant), it requests that the other node perform a corresponding safe operation and waits for notification. (The addressed problem in this protocol was mainly redistributing limits by granting or receiving a slack which is analogous to the quantity exchange problem we address here.) This allowed the propagation of any number of consecutive changes to be made without having to wait for acknowledgments. For these reasons, in addition to its simplicity, the demarcation protocol is still being used nowadays [8], [9], [10], [11]. However, the authors themselves admit that the protocol mis-behaves if no assumptions about message delivery are made. Even though safety is not violated, over time, under message duplication or loss, resources can be "lost" or limits can become overly restrictive, as explained in [6] and [8]. Krishnakumar and Jain tried to avoid these problems in mobile inventory services [6]; however, they used multiple 2PC phases and a third party server, which not only it is very costly but also a single point of failure.

Addressing the problem of reliable communication between two parties, in practice, requires retaining unique message identifiers for the set of received, and delivered, messages at the destination endpoint. Messages can be retransmitted when not acknowledged for some time, and the identifier set in the destination can always filter out received duplicates and ensure exactly-once delivery. The filter set, however, will grow linearly with the number of messages received. In settings that aim for reliable FIFO communication, the long term space requirements in the destination endpoint can be improved to be linear with the number of sources, by storing for each source the number that identifies the last message delivered. Messages received out of order must still be buffered. (Notice that quantity transfers do not necessarily require FIFO, since adding received quantities is commutative.)

Transport layer protocols, such as TCP/IP, try to ensure that only within a connection, data sent from one end-point is delivered *exactly-once* to the other end-point, and in FIFO order [12]. However, if a connection breaks while non-acknowledged sent messages are present, those messages are only guaranteed to be delivered *at-most-once*. To enforce exactly-once, the connection management protocol would have to retain connection specific information between different connection *incarnations* [13], something that TCP/IP avoids [14]. Even weaker properties are provided by UDP, where messages can be lost, duplicated or re-ordered.

Reducing storage requirements at the destination are only possible at the expense of time. Attiya and Rappoport have shown, in [13], that endpoints can retain counters that are not connection specific if *at least* a three-way handshake is used to establish a connection. This incurs a latency cost in the first exactly-once transmission. Since its quite reasonable to expect quantity transfers among geo-distributed data centers that aim to keep local escrow for high-availability, and these will have be connected by high latency links, the three-way handshake is particularly taxing for short lived connections.

In this paper, we leverage the fact that we focus on specific short lived task, *quantity transfer*, to make as much progress as possible in the two initial communications of the three-way handshake and use the third communication step to do the exactly-once transfer. The extra information that is piggy-backed in the initial steps, while important for progress, is not required to be done exactly-once; and thus any undetected duplications do not harm the correctness of the exchange and the conservation of quantities. The protocol keeps extra information during the exchange, but after the transfer occurs the state in a node only stores a globally unique node identifier and one counter (nodes that both send and receive store two counters).

The original idea in this paper is inspired from *Handoff Counters* [15] where the authors design scalable eventually consistent counter CRDTs [16] that can work correctly despite network partitions, and avoid the identity explosion problems of previous CRDTs like G-Counters [16]. However, this paper generalizes this idea to quantity transfers in any "splittable" datatype and also expands the application spectrum of the idea to new possible use-cases.

Given the limited paper space, we present our redistribution protocol addressing a specific use-case: reliably moving quantities from a source host to a destination host. A quantity is a simple abstraction that represents a value that can be split into two values that added back together produces the original value. A simple example is that of money transfer between two wallets. Money in origin wallet o in variable $o.\mathsf{val}$ is split into $o.\mathsf{val}'$ and m, with $o.\mathsf{val} = o.\mathsf{val}' + m$; then m is transferred, exactly-once, to a destination wallet d that changes the stored

$$\begin{aligned} 0 &\doteq 0 \\ \oplus &\doteq + \\ \mathsf{needs}(x,y) &\doteq \frac{y-x+|y-x|}{4} \\ \mathsf{split}(x,h) &\doteq \left(\frac{x-h+|x-h|}{2}, \frac{x+h-|x-h|}{2}\right) \end{aligned}$$

Fig. 1. $\mathbb{R}$ Data type example: Positive reals that ask for half difference (when smaller) and give as much as possible.

amount to $d.\mathsf{val}' = d.\mathsf{val} + m$. No money is lost or created, since $o.\mathsf{val} + d.\mathsf{val} = o.\mathsf{val}' + d.\mathsf{val}'$. The same principle can be applied to many applications: stock escrow, token transfers, service handoffs, etc [17], [6], [18].

In the future, we plan to present this concept more formally, including transfer policies discussions, protocol variants, and empirical experimentation.

II. Protocol

A. System Model

Consider a distributed system with nodes containing local memory, with no shared memory between them. Any node can send messages to any other node. The network is asynchronous, there being no global clock, no bound on the time it takes for a message to arrive, nor bounds on relative processing speeds. The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted). Some messages will, however, eventually get through: if a node sends infinitely many messages to another node, infinitely many of these will be delivered. In particular, this means that there can be arbitrarily long partitions, but these will eventually heal.

The system is composed of n nodes. Nodes have access to stable storage. Nodes can crash but eventually will recover with the content of the stable storage as at the time of the crash. We assume no Byzantine or Rational behaviors.

B. Payload Data Types

Valid data values types T must be commutative monoids with a generic sum operator $\oplus$, and identity element 0. (Splittable data values in the related work history were called partitionable, fragmentable, or even escrowable; in this paper we choose to use commutative monoids as it captures the essential mathematical properties that are actually required.) Fragmenting a quantity is done via a user defined split function; it can be any function such that $(x', q) = \mathsf{split}(x, h) \Leftrightarrow x' \oplus q = x$. Some split functions can use the hint h to further ensure that $0 \leq q \leq h$, but this is not needed for correction. Load-balancing is abstracted via a user defined needs function that compares a local amount to a remote amount, deciding how much to ask. It can be such that $h = \mathsf{needs}(x, y)$ creates a hint h and that typically we will have $0 \leq h \leq y$, with h representing a value that is beneficial to split from y and move to x.

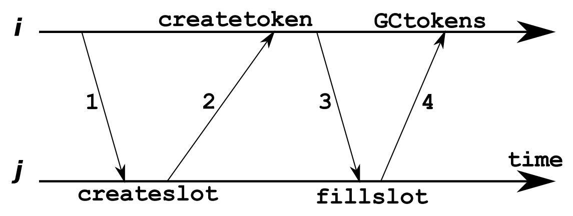


Fig. 2. Basic fault-free communication scenario.

val	data value reported by fetch;
sck	source clock – logical clock incremented when creating tokens;
dck	destination clock – logical clock incremented when creating slots;
slots	map from source ids to pairs $((sck, dck), D)$ containing a pair of logical clocks and a data value;
tokens	map from destination ids to pairs $((sck, dck), D)$ containing a pair of logical clocks and a data value;

Fig. 3. Replica state (record fields)

C. Use-case and Redistribution Policy

Since discussing redistribution policies is not the focus of this short article, we assume the following use-case: a quantity (e.g., a *real* number) that needs to be evenly balanced over all replicas, so that they try to keep similar fractions of the global amount. Replicas periodically share their values; a replica that owns more credits will *split* its value and transfer them to another replica that *needs* it (i.e., has scarce resources). Fig. 1 shows how split and needs are performed over real numbers. In addition, we assume that redistribution occurs in a periodic fashion. We aim at supporting more policies in the future.

While Fig. 1 shows how these datatype-specific functions can be implemented for positive reals, we have also built and tested similar definitions for integers and for maps from identifiers to integers, that would more directly represent stock/inventory abstractions.

D. The Algorithm

Our protocol, running on each node, is depicted in Algorithm 1, and makes use of auxiliary functions defined in Fig. 4.. Each node has access to a globally unique identifier i, a set of neighbors n_i, and an initial quantity v^0 of a valid data type. The algorithm shows operations that can be invoked locally, to act on the local data type; how to handle messages received; and triggers periodic transmissions to other nodes. The node state is a record with fields shown in Fig. 3.

a) Overview: Fig. 2 depicts the basic fault-free communication scenario of our algorithm. Node j receives a (periodically sent) message from i. Node j notices that i has more resources (a larger quantity) and asks for some quantity by creating a *slot* (a receptor) for i. When i eventually receives the message, it checks if it still has extra resources and *splits* its local val, creating a *token* containing the split quantity, and sends a message with the token to j. As soon as j receives the token, it adds the quantity to its val, removing the slot

Algorithm 1: Distributed algorithm for a generic node i.

```
constants:
    $i$, globally unique node id
    $n_i$, set of neighbors of node $i$
    $v^0$, initial data value of type T
state:
    $C_i = \{\mathsf{val} = v^0, \mathsf{sck} = 0, \mathsf{dck} = 0,$
        $\mathsf{slots} = \{\}, \mathsf{tokens} = \{\}\}$
local $\mathsf{fetch}_i$
    return $C_i.\mathsf{val}$
local $\mathsf{plus}_i(q)$
    $C_i := C_i\{\mathsf{val} = \mathsf{val}_i \oplus q\}$
local $\mathsf{minus}_i(q)$
    let $(v, q') = \mathsf{split}(C_i.\mathsf{val}, q)$
    $C_i := C_i\{\mathsf{val} = v\}$
    return $q'$
on $\mathsf{receive}_{j,i}(C_j)$
    $C_i := \mathsf{fillslots}(C_i, C_j)$
    $C_i := \mathsf{createslot}(C_i, C_j)$
    $C_i := \mathsf{GCtokens}(C_i, C_j)$
    $C_i := \mathsf{createtoken}(C_i, C_j)$
periodically
    for $j \in n_i$ do
        let $m = C_i\{$
            $\mathsf{slots} = \{(k, s) \in \mathsf{slots}_i \mid k = j\},$
            $\mathsf{tokens} = \{(k, s) \in \mathsf{tokens}_i \mid k = j\}\}$
        $\mathsf{send}_{i,j}(m)$
```

(function fillslots) and replies back to i. Finally, i can safely garbage collect the token (function GCtokens). We describe the algorithm in more detail in the following.

b) Notation: We use mostly standard notation for sets and maps/relations. A map is a set of (k, v) pairs (a relation), where each k is associated with a single v; to emphasize the functional relationship we also use $k \mapsto v$ for entries in a map. We use $M\{\ldots\}$ for map update; $M\{x \mapsto 3\}$ maps x to 3 and behaves like M otherwise. For records we use similar notations but with $=$ instead of $\mapsto$, to emphasize a fixed set of keys. We use $\lhd$ for domain subtraction; $S \lhd M$ is the map obtained by removing from M all pairs (k, v) with $k \in S$. We use set comprehension of the form $\{x \in S \mid P(x)\}$. The domain of a relation R is denoted by $\mathsf{dom}(R)$, while $\mathsf{fst}(T)$ and $\mathsf{snd}(T)$ denote the first and second component, respectively, of a tuple T. To define a function or predicate by cases, we use **if** X **then** Y **else** Z to mean "Y if X is true, Z otherwise".

c) Local functions: Function fetch returns the val field; operation plus adds an amount to val; operation minus attempts to subtract an amount from val, limited to the available quantity, as val cannot go below zero, returning the amount

```
$\mathsf{fillslots}(C_i, C_j) \doteq$ if $(i, (ck, q)) \in \mathsf{tokens}_j \wedge (j, (ck, \_)) \in \mathsf{slots}_i$
    then $C_i\{\mathsf{val} = \mathsf{val}_i \oplus q, \mathsf{slots} = \{j\} \lhd \mathsf{slots}_i\}$
  else if $(j, ((sck, \_), \_)) \in \mathsf{slots}_i \wedge \mathsf{sck}_j > sck$
    then $C_i\{\mathsf{slots} = \{j\} \lhd \mathsf{slots}_i\}$
  else $C_i$
$\mathsf{createslot}(C_i, C_j) \doteq$ let $h = \mathsf{needs}(\mathsf{val}_i, \mathsf{val}_j)$
  if $j \notin \mathsf{dom}(\mathsf{slots}_i) \wedge h \neq 0$
    then $C_i\{\mathsf{slots} = \mathsf{slots}_i\{j \mapsto ((\mathsf{sck}_j, \mathsf{dck}_i), h)\}, \mathsf{dck} = \mathsf{dck}_i + 1\}$
  else $C_i$
$\mathsf{GCtokens}(C_i, C_j) \doteq$ if $j \in \mathsf{dom}(\mathsf{tokens}_i) \wedge (i \in \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{snd}(\mathsf{fst}(\mathsf{tokens}_i(j))) < \mathsf{snd}(\mathsf{fst}(\mathsf{slots}_j(i)))$
      $\vee\ i \notin \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{snd}(\mathsf{fst}(\mathsf{tokens}_i(j))) < \mathsf{dck}_j)$
    then $C_i\{\mathsf{tokens} = \{j\} \lhd \mathsf{tokens}_i\}$
  else $C_i$
$\mathsf{createtoken}(C_i, C_j) \doteq$ if $i \in \mathsf{dom}(\mathsf{slots}_j) \wedge \mathsf{fst}(\mathsf{fst}(\mathsf{slots}_j(i))) = \mathsf{sck}_i$
    then let $(v, q) = \mathsf{split}(\mathsf{val}_i, \mathsf{snd}(\mathsf{slots}_j(i)))$
        $C_i\{\mathsf{tokens} = \mathsf{tokens}_i\{j \mapsto (\mathsf{fst}(\mathsf{slots}_j(i)), q)\},$
          $\mathsf{val} = v,$
          $\mathsf{sck} = \mathsf{sck}_i + 1\}$
  else $C_i$
```

Fig. 4. Auxiliary functions in receive.

actually subtracted. It makes use of function split that splits val into two amounts.

d) Sending: Periodically, each node i sends a message to each neighbor j, containing the *view* of its state, containing only the information that is relevant to the specific receiver j. Notice that while the connection between two nodes is unreliable, as sending is done periodically, eventually a message will be received. We do not specify a specific network topology, but the algorithm will balance values in each connected component. For simplicity the reader can picture a simple topology with a single connected component, such as a ring or a complete graph.

e) Receiving: Once i receives a new message from another node j, it incorporates it into its state by performing four steps, using the functions from Fig. 4. These functions receive as argument two state records and return the new state, possibly with some of its fields updated.

Node i starts by checking if it has open slots for j and tries to fill them if so (fillslots); it first verifies if j has a token for i (that must have been previously created) and if that very token has a locally opened slot on i (a matching ck). In this case, i adds the received amount q to its val and removes the corresponding slot. On the contrary, if a slot for j exists on i but ck is not matching, i tries to garbage collect the slot if the source clock of j is ahead the clock registered in the designated local slot. This basically means that j has already created a token to another node (and incremented it local clock sck_j) to acquire lacking amounts and discarded creating a token corresponding to a previously sent slot by i.

Then it decides whether it should create a slot for j (createslot); if i has no open slot for j, it opens a corresponding slot only if $h \neq 0$ using the needs function (meaning that j has excess amount to offer to i). Thus, i stores the newly created slot that corresponds to (sck_j, dck_i) and advances its sending clock dck_i. Notice that since this is only done if i has no open slots for j, this guarantees that no slots are created for duplicate messages if sck_j has not been incremented (otherwise garbage collection would have occurred and a new slot creation is allowed).

The next step is to check if node i has a token for j due to a previous contact. In this case, the token may have been successfully merged by j, and thus this token has to be garbage collected GCtokens if: j has no open slots for i and its slots clock dck_j is ahead the said clock of the stored token. The last step is to create a token if j has an open slot for i such that the clock of the slot and node i are matching. In this case, i splits its val using split to hand it off to j. Recall that, split shall not return the exact amount needed by j if val is not large enough according to the policy in Fig. 1).

III. Correctness

We provide an informal proof for the correctness (safety and liveness) of our protocol. We postpone formal proofs to an extended version due to page limits.

A. Safety

We explain safety by focusing on duplicated messages, reordering, and "lost resources" problems since this is the aim of the protocol. We omit the cases of lost message as we assume eventual delivery and we explain re-ordering when needed.

Consider phase 1 in Fig. 2; this phase is safe under message duplication since a duplicate slot will never be created as per the conditions in $\mathsf{createslot}$. In phase 2, upon receiving an open slot, i creates a corresponding token and advances its clock sck_i. Receiving a duplicate slot will have no effect since the slot's clock will not be matching anymore with sck_i. In phase 3, once j receives a token from i it fills the corresponding slot (and deletes it); receiving another duplicate of the same token will have no effect since there is no receptor slot at j. The final phase 4 is also duplication-safe since a token will be garbage collected only once.

The algorithm is also safe against message re-ordering. As depicted on Fig. 2, there are only two re-ordering possibilities: (1) Phase 1 and 3 are re-ordered. This is impossible to occur since there is no way that j creates a slot (and consequently i creates the corresponding token) unless if j received a prior message, i.e., in phase 1. (2) Phase 2 and 4 are re-ordered. This case is safe as i would simply discard the message since it has no matching token for j.

As for "lost resources", the only way for offering resources is to $\mathsf{createtoken}$ (in which split is called); but as explained above, this can only occur if the other node, j, has already a corresponding open slot. In addition, j could not delete a slot unless i's sck is ahead the slot's clock (**else if** in $\mathsf{fillslots}$), which means that i has already sent a token to another node and it could not offer j a quantity; thus j must eventually add (in $\mathsf{fillslots}$) the split quantity (in $\mathsf{createtoken}$) sent in the token from i (phase 3).

B. Liveness

As for liveness, we first recall that we assume eventual delivery of messages across all system nodes. Therefore, network partitions, though possible, are considered transient and messages will eventually go through. Now, we informally demonstrate the liveness of the algorithm using Fig. 2.

First, notice in Fig. 2 that node j can $\mathsf{createslot}$ and $\mathsf{fillslots}$ without blocking. In fact, j can always run $\mathsf{createslot}$ to create a slot for i if its quantity is less than that of i. (An existing slot would have been removed in $\mathsf{fillslots}$.) In addition, j does not have to wait until a token is received from i; however, it could create other slots to other nodes too. Node j can thus remove a created slot only when it receives a matching token as shown in $\mathsf{fillslots}$ function in Fig. 4; otherwise, the slot is kept (until it is eventually garbage collected), which has no impact on progress.

As for node i, it only creates a token in $\mathsf{createtoken}$ if an open slot is received and it still has larger quantity (as it could have transferred some to another node by sending a prior token). This is okay since j will eventually garbage collect the corresponding slot. After creating the token, i will increment its clock sck_i; this prevents it from creating any other token to j unless it has received an ACK (i.e., a new slot with matching sck_i) from it, since the condition $\mathsf{fst}(\mathsf{fst}(\mathsf{slots}_j(i))) = \mathsf{sck}_i$ will not be satisfied in $\mathsf{createtoken}$. However, in all cases, i will be able to create tokens to other nodes in the system if a matching slot (holding the new incremented sck_i) is received and i has extra quantity to transfer. Node i can then send new tokens to j once an ACK from j is received and $\mathsf{GCtokens}$ is applied (which will eventually occur).

Finally, the protocol will not block due to the transfer policy since nodes with larger quantities will always offer quantities to other nodes. This is guaranteed as we assume all nodes are periodically exchanging states even if no local events occurred. This can obviously be done in more efficient ways according to the policy chosen (which we do not discuss here).

IV. Discussion and Future Work

In this paper, we focused on presenting the idea of the algorithm on a simple real number averaging example. However, the reader can easily notice that this algorithm can be used in other cases of similar $split/merge$ nature, as in [17], [5], [19], [8], etc. We described our algorithm keeping in mind an averaging policy whereas multiple policies could by addressed. In this specific policy we did not address if averaging occurs or not, but we rather focused on the correctness of the algorithm. We conducted preliminary empirical evaluations to this averaging problem on up to 1000 nodes and the results seem promising: all nodes started with high variance of quantities and came to an average value, while all meta-data (tokens and slots) were garbage collected. We aim to provide more evaluation and comparison results in the future.

In addition, we assumed that messages are simply disseminated in a periodic fashion (e.g., through gossiping); clearly, other options can be of interest too like having the node with scarce resources ask other nodes (avoiding periodic dissemination). We have also assumed that no transitive sending occurs between nodes, meaning that a third-party node could not deliver a message on behalf of another node. We think that this case is worth more focus in the future.

V. Related Work

The problem of quantity transfer or redistribution (sometimes called repartitioning or reconfiguration) first appeared in the context of database transactions by Carvalho et al. [2] to maintain the invariants (or *limits*) on different servers as in the Escrow Transactional method [20], [17]. The aim was to redistribute an "escrowable" (or fragmentable) value (a limit) over multiple partitions in a distributed storage by "splitting" an amount on one replica and adding it to another. This idea of "splitting" was first proposed by Davidson et al., in [21], inspired by [22] in the context of reliable networks. Another protocol was later proposed in [1] where a node can "borrow" elements from neighbors (and waits) until acknowledged. They used "partitionable operators" (similar to the $\oplus$ operator we use in our paper); however, this protocol had impractical weaknesses like blocking, re-ordering, and duplication.

The famous "demarcation" protocol was then introduced by Barbara et al. in [3], [4]. This protocol aimed at maintaining invariants in distributed databases using escrow-like method [20], [17]. The demarcation protocol was immune to message delays and the order of reception and would allow the propagation of any number of consecutive changes to be made without having to wait for acknowledgments. This was a substantial improvement over its predecessors as it could tolerate network partitions. The protocol however is not immune to network problems like message dropping and duplication which can lead to incorrect behaviors like more conservative limits (in case of limit management) or "lost resources" (in case of quantity transfer as in our work). Several protocols were then proposed by Krishnakumar et al., in [7], [23], [6], in the context of mobile services and inventory to overcome these problems; however, they used many 2PC phases which was not practical for systems that focus on low latency and high availability.

To the best of our knowledge, no further improvements were made to the demarcation protocol, and it is still being used by current systems despite its aforementioned caveats [8], [9], [10], [11].

VI. Conclusion

We introduced a new redistribution protocol to perform an exactly-once transfer of a "quantity" from one node to another in a distributed system. The protocol is immune to delivery problems like message dropping, duplication, and re-ordering. Although this protocol addressed a single "averaging" problem of a distributed *real* number, it is easy to adapt to other contexts, use-cases, and applications. The paper focused on presenting the algorithm and showing its correctness properties leaving other details to a future work, like distribution and splitting policies and other variants of the protocol. Since this is a work in progress, we aim at presenting more formal presentation accompanied with experimentations in a longer version. We already had some promising results showing that the protocol brings all replicas (up to one thousand) to an average value without leaving any garbage traces or meta-data.

VII. Acknowledgment

This work is financed by the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013; and by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 609551, SyncFree project.

References

[1] N. Soparkar and A. Silberschatz, "Data-value partitioning and virtual messages," Austin, TX, USA, Tech. Rep., 1989.

[2] O. S. Carvalho and G. Roucairol, "On the distribution of an assertion," in *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '82. New York, NY, USA: ACM, 1982, pp. 121–131. [Online]. Available: http://doi.acm.org/10.1145/800220.806689

[3] D. Barbará and H. Garcia-Molina, "The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems," in *Proceedings of the 3rd International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT '92. London, UK, UK: Springer-Verlag, 1992, pp. 373–388. [Online]. Available: http://dl.acm.org/citation.cfm?id=645336.649877

[4] D. Barbará-Millá and H. Garcia-Molina, "The demarcation protocol: A technique for maintaining constraints in distributed database systems," *The VLDB Journal*, vol. 3, no. 3, pp. 325–353, Jul. 1994. [Online]. Available: http://dx.doi.org/10.1007/BF01232643

[5] R. Jain and N. Krishnakumar, "Network support for personal information services to pcs users," in *Networks for Personal Communications, 1994. Conference Proceedings., 1994*, Mar 1994, pp. 1–7.

[6] N. Krishnakumar and R. Jain, "Escrow techniques for mobile sales and inventory applications," *Wirel. Netw.*, vol. 3, no. 3, pp. 235–246, Aug. 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1019161318592

[7] N. Krishnakumar and A. J. Bernstein, "High throughput escrow algorithms for replicated databases," in *Proceedings of the 18th International Conference on Very Large Data Bases*, ser. VLDB '92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 175–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=645918.672481

[8] V. Balegas, D. Serra, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Shapiro, and M. Najafzadeh, "Extending eventually consistent cloud databases for enforcing numeric invariants," *CoRR*, vol. abs/1503.09052, 2015. [Online]. Available: http://arxiv.org/abs/1503.09052

[9] A. Elmagarmid, J. Jing, and O. Bukhres, "An efficient and reliable reservation algorithm for mobile transactions," in *Proceedings of the Fourth International Conference on Information and Knowledge Management*, ser. CIKM '95. New York, NY, USA: ACM, 1995, pp. 90–95. [Online]. Available: http://doi.acm.org/10.1145/221270.221338

[10] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: Pay only when it matters," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 253–264, Aug. 2009. [Online]. Available: http://dx.doi.org/10.14778/1687627.1687657

[11] 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*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 113–126. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465363

[12] P. Helland, "Idempotence is not a medical condition," *Queue*, vol. 10, no. 4, pp. 30:30–30:46, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2181796.2187821

[13] H. Attiya and R. Rappoport, "The level of handshake required for establishing a connection," in *Distributed Algorithms*, ser. Lecture Notes in Computer Science, G. Tel and P. Vitnyi, Eds. Springer Berlin Heidelberg, 1994, vol. 857, pp. 179–193. [Online]. Available: http://dx.doi.org/10.1007/BFb0020433

[14] R.Braden, "Tcp extensions for transactions," *RFC*, Jul. 1994. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1644.txt

[15] P. S. Almeida and C. Baquero, "Scalable eventually consistent counters over unreliable networks," *CoRR*, vol. abs/1307.3207, 2013. [Online]. Available: http://arxiv.org/abs/1307.3207

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050613.2050642

[17] A. Kumar and M. Stonebraker, "Semantics based transaction management techniques for replicated data," *SIGMOD Rec.*, vol. 17, no. 3, pp. 117–125, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/971701.50215

[18] Paulo S'ergio Almeida and Ali Shoker and Carlos Baquero., "Efficient State-based CRDTs by Delta-Mutation," in *Proceedings of the International Conference of Networked sYStems*, ser. NETYS'15. Springer, May 2015.

[19] M. Mouly and M.-B. Pautet, *The GSM System for Mobile Communications*. Telecom Publishing, 1992.

[20] P. E. O'Neil, "The escrow transactional method," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 405–430, Dec. 1986. [Online]. Available: http://doi.acm.org/10.1145/7239.7265

[21] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a partitioned network: A survey," *ACM Comput. Surv.*, vol. 17, no. 3, pp. 341–370, Sep. 1985. [Online]. Available: http://doi.acm.org/10.1145/5505.5508

[22] M. Hammer and D. Shipman, "Reliability mechanisms for sdd-1: A system for distributed databases," *ACM Trans. Database Syst.*, vol. 5, no. 4, pp. 431–466, Dec. 1980. [Online]. Available: http://doi.acm.org/10.1145/320610.320621

[23] N. Krishnakumar and R. Jain, "High throughput escrow algorithms for replicated databases," in *Proceedings of the MOBIDATA Workshop*, ser. MOBIDATA '94. Rutgers Univ., 1994.

C Papers accepted for publication

C.1 Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free Partially Replicated Data Types. In Proc. 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015), Vancouver, BC, Nov. 30-Dec. 3, 2015. (to appear)

Conflict-free Partially Replicated Data Types

Iwan Briquemont[*], Manuel Bravo[*†], Zhongmiao Li[*†] and Peter Van Roy[*]
[*]Université catholique de Louvain, Belgium
[†]Instituto Superior Técnico, Universidade de Lisboa, Portugal

***Abstract*—Designers of large user-oriented distributed applications, such as social networks and mobile applications, have adopted measures to improve the responsiveness of their applications. Latency is a major concern as people are very sensitive to it. Geo-replication is a commonly used mechanism to bring the data closer to clients. Nevertheless, reaching the closest datacenter can still be considerably slow. Thus, in order to further reduce the access latency, mobile and web applications may be forced to replicate data at the client-side. Unfortunately, fully replicating large data structures may still be a waste of resources, specially for thin-clients.**

We propose a replication mechanism built upon conflict-free replicated data types (CRDT) to seamlessly replicate parts of large data structures. The mechanism is transparent to developers and gives improvements without increasing application complexity. We define partial replication and give an approach to keep the strong eventual consistency properties of CRDTs with partial replicas. We integrate our mechanism into SwiftCloud, a transactional system that brings geo-replication to clients. We evaluate the solution with a content-sharing application. Our results show improvements in bandwidth, memory, and latency over both classical geo-replication and the existing SwiftCloud solution.

***Keywords*—CRDTs, partial replication, caching**

I. Introduction

Globally accessible web applications, such as social networks, aim to provide low-latency access to their services. Thus, data locality is a fundamental property of their systems. Geo-replication is a common solution where data is replicated in multiple datacenters [1]–[3]. In this scenario, user requests are forwarded to the closest datacenter. Therefore, the latency is reduced. Unfortunately, the latency, even when accessing the closest datacenter, may still be considerable. [4], [5] argue that clients are sensitive to even small increases of latency.

Systems such as [6], [7] use caching techniques to yet reduce latency even more. However, this can be challenging and expensive. For instance, one could simply use client caches for reading purposes. Nevertheless, in order to keep some consistency guarantees and freshness of data, mechanisms, such as cache invalidation, need to be used. Scaling these kinds of techniques is difficult and directly affects the performance. Moreover, one could let clients apply write operations locally and eventually propagate them. However, this can cause conflicts between replicas and potential rollbacks.

The recently formalized CRDTs [8], [9] can serve to diminish the impact of some of the previously mentioned problems. These data types are conflict-free by default; therefore, no conflict resolution mechanisms need to be written by application developers. SwiftCloud [10], a geo-replicated storage system that ensures causal consistency, benefits from CRDT semantics. It replicates CRDTs not only across datacenters, but it also replicates them in clients. It allows read and write operations to be directly executed in client caches. In consequence, SwiftCloud reduces latency, and enables off-line mode during disconnection periods.

The current specifications of CRDTs do not allow partitioning. Thus, a CRDT replica is assumed to contain the full data structure. We believe partitioned CRDTs may pose several benefits for current applications. First, CRDTs can easily become heavy data structures, such as a set CRDT that contains the posts of a user wall in a Facebook-like application. In many cases, users are simply interested in the most relevant posts, according to some criterium. For instance, one may be interested in reading the top-ten most voted posts of a Reddit-like application. Thus, replicating the whole CRDT is a waste of resources, of both storage and bandwidth. The former can be critical when thin devices, such as smartphones, are considered as clients. These types of clients have limited memory resources; therefore, it is convenient to avoid storing unnecessary data. On the other hand, bandwidth is one of the most costly resources offered by cloud providers such as Amazon S3 [11], Google Cloud Storage (GCS) [12], and Microsoft Azure [13]; therefore, it is beneficial to use it efficiently. Second, the full replication of CRDTs in clients may arise security concerns. By partitioning CRDTs, applications could precisely decide which data each client stores. This could keep malicious clients from storing sensitive data. Finally, they can also be used to provide multiple fidelity requirements for data accommodated in resource-limited devices, while keeping consistency between fidelity levels [14]. This could be achieved by not replicating less important information on mobile devices.

In this paper, we propose a new kind of CRDTs that allows partitioning. We call them "Conflict-free Partially Replicated Data Types" (hereafter CPRDTs). We study how partitions of the same CRDT can interact among each other and still maintain its consistency guarantees. Furthermore, we revise previously defined CRDT specifications and propose new specifications that consider partitioning. One could claim that developers could simply re-format their data structures to obtain similar benefits; nevertheless, this adds complexity to application development and, in some cases, optimal results can be difficult to achieve. We propose to better integrate CPRDTs into the system. Thus, developers will benefit from them transparently, without being aware of their existence.

The major contributions of this paper are the following: (i) definition of the new CPRDTs, which includes revisiting the specifications of previously defined CRDTs; (ii) extension of SwiftCloud to integrate CPRDTs; and (iii) extensive evaluation of the performance improvements of CPRDTs in SwiftCloud. The latter includes the implementation of a Reddit-like [15], [16] application, called SwiftLinks, on top of SwiftCloud.

The remainder of the paper is organized as follows: Section II presents a formal definition of the partitioned CRDTs; Section III discusses some practical issues of CPRDTs; Section IV presents an extensive evaluation of the SwiftCloud extension that includes CPRDTs; Section V briefly describes preceding related work; finally, Section VI concludes the paper.

II. Conflict-free Partially Replicated Data Types

Allowing partitioning poses new challenges: all operations are not enabled on partial replicas, which means new preconditions must be added to ensure correctness. However, these preconditions must not compromise the convergence of replicas. Plus, a partial replica could vary the parts it keeps, by choosing to replicate more, or less, parts. This has to be done without losing data and still achieving convergence.

A. Example of use

Let us use an example to illustrate the advantages of CPRDTs: the user wall of a social network. We can model a user's wall as a set. In this example, there are four users that interact: Alice, Bob, Charlie and an anonymous user. Bob is a friend of Alice, while Charlie is a friend of Bob, but not of Alice. Participating users may want to read or post something in Alice's wall. We make two assumptions: (i) users maintain a full replica of their wall; and (ii) a user X that reads or posts in user Y's wall replicates user Y's wall locally.

Each post contains a date, an author, and a message. Each user is allowed to read a subset of other users' walls, depending on their friendship and posts visibility (private or public). For instance, Bob can read all the posts of Alice's wall because of their friendship. Nevertheless, Charlie can only read public and Bob's posts (friends of friends). Finally, any other user can only read public posts.

We can assume that Alice has been using the social network for a few years and there are a considerable number of posts on her wall. It seems natural that a user should not have to replicate the whole wall to simply read the latest posts. Nevertheless, this is what presumably may occur in a fully-replicated scenario (CRDT-like), where the data structures cannot be partitioned and we still want to replicate data in clients-side. One solution is to manually split the data structure according to some criteria (e.g. by date, author or privacy setting). However, developers then need to anticipate how users will use the application. While possible in some cases, it seems to make the application cumbersome to write.

In this scenario, CPRDTs have two applications. On the one hand, CPRDTs abstract the partitioning from the application. Thus, from the point of view of programmers, there will only be one logical data structure per wall. This simplifies the developer's task. Moreover, this allows a more efficient and fine-grained partitioning adapted to the needs of a particular client in a specific point of time, which may be impossible if the partitioning is done manually by developers. The second application of CPRDTs is related to the enforcement of security policies. We may want users to only replicate posts that they are allowed to see. This could keep malicious users from storing sensitive data locally.

B. Definitions

Before defining CPRDTs, we have to clarify some concepts that we will use throughout the paper. An *object* is a named instance of a CRDT or CPRDT in our case. Each participating process replicates a set of objects. A process that replicates an object is called *replica* of the CRDT (or CPRDT) instantiated by the object. Objects can be read using *query* operations and modified by *update* operations. Query operations return the abstract state of the object, that we call the *data* of the object. Nevertheless, additional data, which we refer as *metadata*, is kept internally to ensure convergence.

An update operation can have preconditions that capture its safety requirements. In consequence, an operation is said to be *enabled* at a replica, if it satisfies its preconditions. For instance, the remove operation of a set is enabled only if the element to be removed is present in the set.

Previous definitions fit into both CRDTs and CPRDTs. Nevertheless, for CPRDTs, we further consider that a process might replicate an object partially: it only has access to a part of data, thus the process only keeps the metadata required for that given part. Intuitively, this means that only part of the data structure is replicated: some elements of a set, a subgraph of a graph, or a slice of a sequence. CRDTs that only have one element, such as counters and registers, cannot be partitioned and therefore do not need to be specified as CPRDTs.

particle We define a *particle* as an element of a collection. For instance, a particle in a set would be any element that can be added to the set.

Apart from the definition of particle, we introduce three new concepts: *shard set*, *required*, and *affected*.

shard set Each replica x_i of a CPRDT has associated a set of particles, namely *shard set* in analogy to the databases concept. Respectively, $\mathrm{shard}(x_i)$ is a function that returns the *shard set* of x_i. A replica x_i only knows the state of the particles in $\mathrm{shard}(x_i)$; therefore, it can only enable query operations that require those particles. Furthermore, a replica x_i only needs to receive update operations that affect the particles in $\mathrm{shard}(x_i)$.

There are two special cases: a *full* replica and a *hollow* replica. We use π to represent the full set of possible particles a CPRDT may be interested in. The set π may be infinite. Thus, we say that a *full* replica's *shard set* is equal to π. Notice that a *full* replica CPRDT is equivalent to a normal CRDT. On the other hand, when $\mathrm{shard}(x_i) = \varnothing$, then x_i is a *hollow* replica (as named in [17]). A *hollow* replica does not maintain any state. Nevertheless, it can still handle updates (section II-C2).

required For an operation op with its arguments, $\mathrm{required}(op)$ is the set of particles needed by op to be properly executed. This means that, for replica x_i, an operation op is enabled only if $\mathrm{required}(op) \subseteq \mathrm{shard}(x_i)$. For instance, for the lookup operation of a set, $\mathrm{required}(\mathit{lookup}(e)) = \{e\}$ where e is an element of the set. In case $e \notin \mathit{shard}(x_i)$, the replica x_i does not know whether e is in the set because it has not kept a state for it. This implies that updates affecting e have not been necessarily seen by x_i.

affected The function $\mathrm{affected}(op)$ returns a particle that may have its state affected after executing an update operation. We assume that an update can only affect one particle. This may not be true for complex data structures, however it is always possible to split an operation into several ones that each only affects one particle. For example, for a graph, an operation for removing a vertex will remove the vertex as well as all its edges. It can be split into several sub-operations that firstly remove all edges of the vertex and then remove the vertex.

C. Replication

As for the original CRDTs, we consider two equivalent replication techniques: state- and operation-based. Allowing partitioning introduces changes in the way these replication techniques work. Furthermore, concepts such as causal history and convergence have to be revisited. The following definitions are based on the ones in [8] for fully-replicated CRDTs.

To simplify our definitions, we assume that the *shard set* of a CPRDT is fixed. However, in practice, it can be necessary to dynamically change it. Nevertheless, definitions apply if we consider that changing the *shard set* is equivalent to the creation of a new CPRDT replica.

Since the abstract state of a CPRDT may change after applying an update, we denote the abstract states of a CPRDT replica (x_i) by an increasing numbered sequence as $s_k(x_i)$, such as $s_0(x_i), s_1(x_i)$... $s_k(x_i)$...

Now we define when two replicas are equivalent.

Definition 1 (Equivalence)**.** *x_i and x_j have equivalent abstract states if all `query` operations q, for which* $\text{required}(q) \subseteq (\text{shard}(x_i) \cap \text{shard}(x_j))$*, return the same values.*

Different replicas of the same CPRDT might have different *shard sets*. Thus, we define intersecting abstract state as the abstract state for the particles in the intersection of *shard sets*.

Definition 2 (Intersecting abstract state)**.** *For a replica x_i with its current state $s_k(x_i)$, $s_k(x_i|x_j)$ denotes the s_k state for particles* $\in \text{shard}(x_i) \cap \text{shard}(x_j)$.

The requirement for replicas to converge is that they apply, directly or indirectly, the same update operations. We can informally define the causal history of a replica, denoted by $C_k(x_i)$, as a set containing the applied update operations. As x_i applies each operation, its causal history goes through a sequence of states $C_0(x_i), C_1(x_i), ..., C_k(x_i), ...$ We also define the intersecting causal history as $C_k(x_i|x_j) = \{f \in C_k(x_i)|\, \text{affected}(f) \in (\mathit{shard}(x_i) \cap \mathit{shard}(x_j))\}$. Intuitively, it includes updates from $C_k(x_i)$ that affect the particles of x_j.

Now, we are ready to formally define convergence in the context of CPRDTs:

Definition 3 (Eventual Convergence of Partial Replicas)**.** *Two partial replicas x_i and x_j of an object x converge eventually if the following conditions are met:*

- *Safety: $\forall i, j : \forall k, k'$, if $C_k(x_i|x_j) = C_{k'}(x_j|x_i)$, then $s_k(x_i|x_j) = s_{k'}(x_j|x_i)$.*
- *Liveness: $\forall i, j : \forall k$, if $f \in C_k(x_i)$ and* $\text{affected}(f) \in \text{shard}(x_j)$*, then $\exists k'$ that $f \in C_{k'}(x_j)$.*

1) State-based partial replication: In this replication technique, a replica ships its whole internal state to the rest. Upon arrival, replicas merge both the local and the received states. The merge method is an idempotent, commutative and associative operation that has two replicas internal states as arguments. In the CPRDTs context, the *merge* method used by a replica must only merge the state of the particles belonging to the intersection between local and remote replicas *shard sets*, and ignore the rest.

State-based replication is an interesting propagation mechanism since it poses almost no communication requirements. Nevertheless, it may be expensive to always ship the full internal state. CPRDTs can optimize this technique since only parts of the state need to be sent and received. We define the causal history of a replica for state-based replication as follows:

Definition 4 (Causal History on Partial Replicas - state-based)**.** *For any replica x_i of x:*

- *Initially, $C_0(x_i) = \varnothing$.*
- *Before executing `update` operation f,*
  if $\text{affected}(f) \in \text{shard}(x_i)$ *then execute f and $C_{k+1}(x_i) = C_k(x_i) \cup \{f\}$,*
  otherwise $C_{k+1}(x_i) = C_k(x_i)$.
- *After executing merge against states x_i, x_j, $C_{k+1}(x_i) = C_k(x_i) \cup \{f \in C_{k'}(x_j)|\, \text{affected}(f) \in \text{shard}(x_i)\}$*

To achieve convergence with state-based replication on partial replicas, updates operations cannot be applied if it affects a particle that is not in that replica's *shard set*. This would violate the liveness property of convergence as that update might not be added to the causal history of another replica when merging. Thus, an operation f is disabled if $\text{affected}(f) \notin \text{shard}(x_j)$. On the other hand, since the replicas only converge on their common parts, a replica x_i just needs to send to another, x_j, the state of the intersection of their shards $(\text{shard}(x_i) \cap \text{shard}(x_j))$.

2) Operation-based partial replication: As with classical CRDTs, the update operations are divided into two phases: *prepare* and *downstream* phase. The former is done at the source replica and does not have any side-effect. The latter is applied at all replicas and it affects the state of replicas. We define the causal history of a replica for operation-based replication as follows:

Definition 5 (Causal History on Partial Replicas - op-based)**.** *For any replica x_i of x:*

- *Initially, $C_0(x_i) = \varnothing$.*
- *After executing the `downstream` phase of operation f at replica x_i,*
  if $\text{affected}(f) \in \text{shard}(x_i)$ *then* $C_{k+1}(x_i) = C_k(x_i) \cup \{f\}$,
  otherwise $C_{k+1}(x_i) = C_k(x_i)$.

In contrast to CRDTs, CPRDTs only have to broadcast updates to the replicas interested in the particles affected by the update. Therefore, an update u is broadcasted to x_i if $\text{affected}(u) \in \text{shard}(x_i)$. This poses an interesting situation.

A CPRDT replica can sometimes complete the first phase of the update operation without necessarily completing the second phase. For instance, a replica x_i, whose $\mathrm{shard}(x_i)$ are particles a and b, receives an update operation that affects particle c. In this situation x_i may complete the prepare phase, broadcast the downstream operation to the interested replicas, and discard it locally. We name this scenario *blind updates*. This can only happen in operation-based replication. Hollow replicas, whose shard is empty, can only do blind updates.

D. Specification of CPRDTs

In this section, we present the specifications of an operation-based observed-remove set (OR-set) CPRDT. We resort into this example in order to better illustrate how to integrate the newly defined concepts into a CRDT (original specifications in [8]); and thus, transform it into a CPRDT. More examples of CPRDTs and generic specification templates, for both operation- and state-based, are found in [18].

An OR-set works as follows: (i) elements are uniquely tagged by the source replica when added to the set. The source replica is the one receiving the client operation. (ii) concurrent additions of the same element are all reflected in the set internal state by storing them with different tags. (iii) a remove operation is transformed into the list of unique tags related to the element to be removed that are present in the source replica. Since causal delivery is assumed, this ensures convergence of replicas even in the presence of concurrent adds and removes of the same element.

The specifications incorporate (i) the particle definition (line 1); (ii) the *required* and *affected* preconditions (lines 11, 15 and 19); and (iii) a new function called *fraction*. The *fraction* operation allows us to create new partial replicas from a subset of a given replica. The subset we want to copy in the new replica is defined by a set of particles. More formally, *fraction* can be defined as follows:
$x_j = \mathrm{fraction}(x_i, Z)$, where Z is the set of particles to replicate. The operations ensures that $\mathrm{shard}(x_j) = \mathrm{shard}(x_i) \cap Z$.

Specification 1 Op-based OR-set with Partial Replication

```
1:  particle definition A possible element of the set.
2:  payload set $S$
3:      initial $\varnothing$
4:  query lookup(element $e$) : boolean $b$
5:      required particles $\{e\}$
6:      let $b = \exists u : (e,u) \in S$
7:  update add(element $e$)
8:      prepare $(e)$ : $\alpha$
9:          let $\alpha = unique()$
10:     effect $(e,\alpha)$
11:         affected particles $\{e\}$
12:         $S := S \cup \{e,\alpha\}$
13: update remove(element $e$)
14:     prepare $(e)$ : R
15:         required particles $\{e\}$
16:         pre lookup($e$)
17:         let $R = \{(e,u) | \exists u : (e,u) \in S\}$
18:     effect $(R)$
19:         affected particles $\{e\}$
20:         pre $\forall (e,u) \in R : add(e,u)$ has been delivered
21:         $S := S \setminus R$
22: fraction (particles $Z$) : payload $D$
23:     let $D.S = \{(e,u) \in S | e \in Z\}$
```

III. Practical issues

In this section, we discuss (i) *shard queries*, and (ii) the implications of allowing dynamic shard sets. Both issues are relevant for making CPRDTs practical.

A. Shard queries

The operation *fraction*, introduced in II-D, is the canonical form to define the *shard set* of a replica. Nevertheless, *fraction* is not practical. In practice, applications will transform their semantics into a high-level query language. For instance, an application could issue a query in the form of "give me the first 10 elements of your sorted set". We name this type of queries *shard queries*. They bridge the gap between the application semantics and the function *fraction* adding expressiveness to the usage of CPRDTs.

There are two types of *shard queries*: version-independent and version-dependent. The former only depends on the properties of the particles, and not in the version of the CPRDT. In contrast, the latter depends on the current version of the CPRDT. Let us use a CPRDT set whose domain is the set of integers as example. A version-independent query could be "integers greater than 0". This *shard query* will always return the same *shard set* $((0, +\infty))$ independently of the queried CPRDT version. On the other hand, a version-dependent query, such as "the 10 highest integers in the set", will return a different *shard set* depending on which elements have been already added, and removed, on the version being queried.

Version-independent queries are easier to work with: they are comparable. One could determine which query is more specific without knowing the version of the CPDRT they apply to. While with version-dependent queries, one can only compare queries if they apply to the same version. Nevertheless, both types are needed in order to make CPRDTs practical.

B. Dynamic shard set

Dynamic *shard set* refers to the capability of a partial replica to modify, either shrink or expand, its *shard set*. We believe this capability is useful in practice, e.g. a client may become interested in new parts. Having dynamic *shard set*, a replica does not need to be re-created, only the missing state needs to be grabbed. Nevertheless, maintaining convergence in some scenarios can become challenging.

On the one hand, a partial replica can easily shrink its *shard set* without compromising convergence in the operation-based scenario. A replica only needs to take two things into consideration: (i) updates prepared locally have been already broadcasted, and (ii) the data to be dropped is replicated by some other replica; therefore, data is not lost. On the other hand, expanding a partial replica is more tricky. For instance, in an operation-based scenario, the following situation can easily occur: (i) a replica's (x_i) *shard set* is a, c; therefore, x_i does not receive updates that affect b; (ii) suddenly, x_i becomes interested in b and starts accepting updates on b; (iii) unfortunately, x_i will not converge since updates have been missed. In order to ensure convergence, extra communication between replicas would be needed to recover dropped updates. This would add complexity to the underlying protocols.

In state-based replication, shrinking or expanding the *shard set* is simpler. On the one hand, a replica only needs to broadcast its state before shrinking its *shard set*. On the other hand, a replica that wants to expand its *shard set* only needs to merge its current state with the state of a replica that contains new particles.

IV. Evaluation

In this section, we report the results of our experimental evaluation. This study aims at evaluating the benefits of CPRDTs in terms of memory, bandwidth and latency.

SwiftLinks In order to evaluate our solution, we implemented an application, namely SwiftLinks, on top of SwiftCloud. SwiftLinks is a vote-based content-sharing application based on Reddit. In short, the application allows users to create forums where they can publish posts. Then, users can vote posts positively or negatively. As a consequence, posts get ranked. In addition, users can comment posts and other comments. Users can also vote comments, and consequently, comments get ranked (more information [15], [16]).

SwiftLinks is modeled with three types of data structures: (i) OR-Set for each forum, (ii) a novel Remove-once Tree for each tree of comments, and (iii) Last-Writer-Wins Registers for each vote associated to a post/comment. The application uses both types of queries: version-independent and version-dependent. The former is mostly used for reading single comments or posts. The latter is used for reading ranking of posts and comments.

Warm-up We used Reddit's API to fetch data to warm up our system. For each benchmark, we create 10000 posts over 20 forums (so an average of 500 posts per forum). Each post has 20 comments on average. Moreover, each post has an average of 170 votes, while comments an average of 13 votes.

Workload Our workloads are composed by read and update operations. Read operations are executed over posts and comments. On the other hand, there are three types of update operations: (i) new post, (ii) new comment, and (iii) new vote.

For most of the experiments, 20% of the operations are updates and 80% are reads. Furthermore, 90% of the operations are biased to previously accessed objects. This means that they are likely to hit the cache. The rest (10%) is done on randomly selected posts and comments.

A. Integration of CPRDTs into a real system

We chose SwiftCloud [10] to integrate CPRDTs. SwiftCloud is a geo-replicated cloud storage system written in Java that stores CRDTs and caches data at clients. It consists of several datacenters that fully replicate the key-space. Clients indirectly communicate through the datacenters. In absence of failures, a client always interacts with its closest datacenter and caches accessed data in its local cache. SwiftCloud provides transactional causal+ consistency. Transactions are first executed and committed on the client side, then propagated to the datacenters. For fault tolerance purposes, committed transactions are only visible after they have been seen by K datacenters.

In our version of SwiftCloud, datacenters store full replicas as in the original implementation. Nevertheless, clients only cache partial replicas. Having full replicas coexisting with the partial replicas considerably simplifies the management of the latter. This poses several advantages in comparison to an ad-hoc architecture where no full replicas, namely authorities, are assumed. Firstly, clients can discard their (partial) replicas at will as long as their updates have been reliably sent to an authority. Secondly, a client can request any fraction from an authority in order to either get a new partial replica, or to expand its own *shard set*. Notice that having an authority also simplifies the integration of state-dependent *shard queries* in the system, very difficult and costly otherwise. Finally, the authority could store which particles each partial replica has in his *shard set*. Thus, it could only propagate operations to the interested replicas, saving bandwidth.

B. Experimental setup

SwiftLinks was evaluated using three Amazon EC2 servers as datacenters: one in Ireland and two in the USA (east and west coast). The EC2 instances are equivalent to a single core 64-bit 2.8 GHz Intel Xeon virtual processor (4 ECUs) with 7.5 GB of RAM. The clients run in 15 PlanetLab nodes located near the DCs. These nodes have heterogeneous configurations with varying processing power and RAM. We set up five SwiftLinks users running concurrently per node, a total of 75. Each client performs an operation per second.

Throughout the evaluation, we use three different modes:

- *Cloud*: This mode simulates a typical geo-replicated system. Clients do not cache any data. Operations are applied synchronously at one datacenter and asynchronously replicated to the rest of datacenters.
- *Partial*. This is the mode that integrates the CPRDTs. Thus, clients only fetch and cache parts of the data structures (CRDTs) as needed.
- *Full*. This is the SwiftCloud approach. Clients cache whole CRDTs even when only part of it is needed.

We limit the capacity of the cache in our experiments to simulate memory restrictions on thin clients, such as a mobile phone. Nowadays, a mobile phone can have up to several gigabytes of memory, but it can easily have tens of applications running simultaneously. An application needs to cohabit with many other applications with limited memory. Therefore, we use 64MB as the default size for cache. If the cache size exceeds this limit, the least recently used object is dropped. In this configuration, *full* and *partial*, if the cache contains the required data, the operations are run locally, and asynchronously propagated to the closest datacenter.

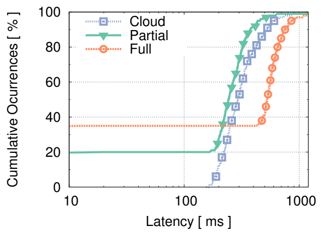
The difference between *full* and *partial* is that the latter benefits from the partial replication mechanism described in the paper. This means that objects are fetched in parts as needed, so the cache can hold parts of an object. For instance, a query for the top ten posts of a forum would only replicate those ten posts in clients cache. On the other hand, for the *full* mode, the objects are only fully replicated in clients side, as in SwiftCloud. Therefore, the same top ten posts query would replicate the whole forum.

C. *Latency*

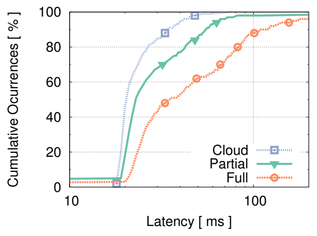
We evaluated the perceived latency for various operations with and without partial object replication. Figure 1 shows the cumulative distribution functions of different operations' latency. These results are obtained after a warm-up phase for the cache. This means that the cache is pre-filled with objects that will be used by the operations present in the workload. For the *full* and *partial* mode, there are always a percentage of operations with a very low latency. We can conclude that it is the percentage of operations that hit the cache.

Read operations Figure 1a shows that the *full* mode has greater cache hit rate (35%) than the *partial* mode. Nevertheless, the hit rate is not optimal due to the limit in the cache size: the cache cannot hold full replicas of all the forums and thus sometimes need to fetch them again. Figure 1c shows the results of a similar experiment but without any cache size limit. In that case, the cache hit rate, for the *full* mode, is 90%, which corresponds to our ratio of biased operations, and it confirms the previous results with a social network application of the SwiftCloud paper [10]. On the other hand, in *partial* mode, the cache hit rate is lower, with only 20% in both experiments (figures 1a and 1c), because the cache only holds partial replicas which gives it less chance of having all the parts needed for hitting the cache in subsequent operations. However, it has the advantage of a lower maximum latency: if an operation does not hit the cache, it only needs to fetch some parts, instead of the full object. In that scenario, it induces a delay similar to the cloud solution, around 200 to 300 ms, while without partial replication, the delay is increased to around 500 to 700 ms by having to replicate a full object. This poses a trade-off between the cache hit rate and the maximum latency. While fully replicating an object will provide more cache hits, a cache miss is more costly.

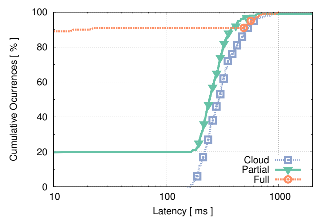
For the latency of reading comments of a post, shown in Figure 1b, the situation is a bit different. Clients are less likely to read the same comment tree multiple times; therefore, this affects the cache hit ratio. As the figure shows, the hit ratio is less than 5% in both *partial* and *full* replication. But again, *partial* replication has the advantage of reducing the impact of a cache miss as it only replicates the comments required by the operation instead of the full comment tree. In consequence, the *partial* approach has a slightly better latency, close to the *cloud* mode. The *cloud* mode performs better because it never needs to fetch any extra metadata, which means that the returned messages are considerable smaller. Notice that the difference between *full* and *partial* mode has been reduced in this experiment because the involved objects are smaller.



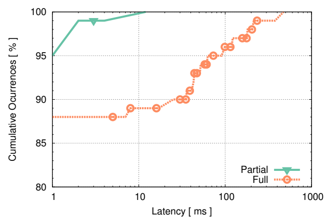
(a) Reading posts. 64MB cache.



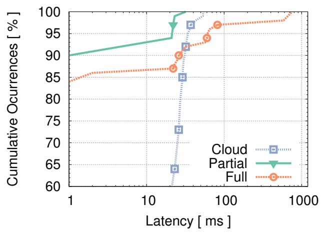
(b) Reading comments. 64MB cache.



(c) Reading posts. Unlimited.



(d) Commenting. 64MB cache.



(e) A mixed update-only workload of posting, commenting, voting a post, and voting a comment

Fig. 1: CDF of SwiftLinks operation latencies

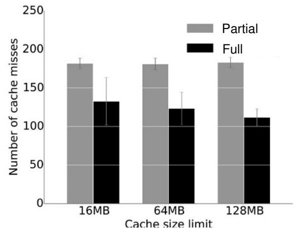
Update operations Caching modes (*full* and *partial*) are more beneficial with update operations. The reason is that update operations are typically applied on objects, or parts of objects, that have already been read by the client. In addition, the update operations only use version-independent queries to fetch their missing parts, which substantially simplifies the comparison of partial objects in the cache. Figure 1e proves experimentally our reasoning. While the cloud mode has an almost constant latency for all operations of a round-trip time, with caching modes, most of the operations (almost 90%) have no latency. Again, the *partial* mode has the advantage of reducing the latency when the cache is not hit, as it only needs to fetch the parts of the object that need to be updated, instead of the full object. Moreover, some updates can be done blindly, therefore, they are completed locally.

In particular, Figure 1d shows the benefit of updates when posting comments, which almost always only requires particles already present in the cache. One can see that with partial replication, all the operations have almost no latency, as they can be done completely asynchronously. In contrast, in *full* mode, there can be a large delay when the tree of comments is not in the cache, as it needs to be fetched from the store. As in previous scenarios, even if an operation cannot be done completely locally in *partial* mode, the client only has to fetch part of the tree to complete the update.

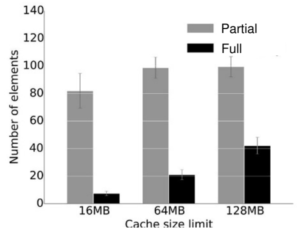
D. *Impact of cache size limit*

In this section we look at how the application performance changes with various cache size limits (16, 64, and 128MB).

1) Impact on latency: We have empirically demonstrated that the *partial* mode performs better without cache limit when reading links. We run the same experiments showed in figures 1a, 1b and 1e setting the cache size limit to 16MB and 128MB. The experiments show that a smaller cache (16MB) size limit has a big latency impact on reading links and updates in *full* mode. Nevertheless, its impact is considerable smaller in



(a) Number of cache misses with different cache size limits.



(b) Total number of objects kept in the cache.

Fig. 2: Impact of cache size limit in partial and full modes

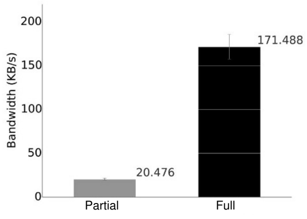
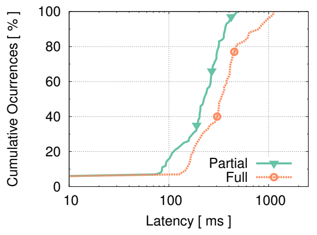
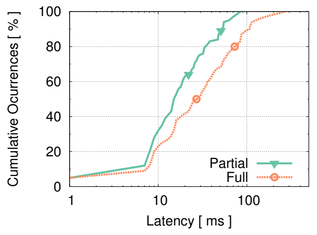


Fig. 3: Average bandwidth usage to fetch objects with a 128MB cache limit, with the cache already warmed up.



(a) Reading posts



(b) Reading comments

Fig. 4: Perceived latency of SwiftLinks during cache warm up.

partial mode. With a small cache, the cache hit rate of *full* mode of reading links becomes worse than in *partial* mode. This is because only a few objects can fit in the cache at a given time; therefore, clients need to fetch objects more frequently. This results in a lower fraction of operations having no latency, about 5% against the 35% obtained with a 64MB cache. There is also an impact for the *partial* mode, but it is considerable lower: it only drops to 13% from 20%. The same applies for update operations. Nevertheless, reads of comments are almost not impacted by the cache size limit: the operations have a low cache locality, so most operations need to fetch an object from the datacenter.

With a 128MB cache size limit, the *full* mode has a large portion of zero latency operations when reading posts, as more are kept in the cache. It however still performs worse than *partial* fetching for operations that do not hit the cache. The latency of updates also improves for the *full* mode with larger cache size, but the *partial* mode still outperforms it.

2) Impact on cache miss rate: The size limit imposed on the cache also has an impact on the cache hit rate. Figure 2a shows that the *partial* mode is less impacted by the cache size limit than the *full* mode. With the three cache limits, the *partial* mode shows a rather stable number of cache misses, about 180. Nevertheless, this does not apply to the *full* mode, where the number of caches misses increases as the cache size is reduced. As in previous experiments, the cache miss rate is greater in the *partial* mode. Nevertheless, we have shown that latency in *partial* mode, is always smaller in average.

3) Impact on number of objects in the cache: The cache size also impacts the number of objects that can be kept in the cache. Notice that for partial replication, only one object is counted even if multiple parts of it have been fetched over time. Figure 2b shows the difference between both modes: *partial* and *full*. In the *partial* mode, many more objects can fit in the cache at any moment, since only parts are kept. 64MB is enough to keep all the objects needed by the application, while in the *full* mode, even 128MB is not enough. This, depending on the workload, may increase the cache hit rate.

E. Bandwidth usage

In *partial mode*, when a client accesses an object, only the needed part of that object is fetched. This can result in saving bandwidth usage compared to *full* mode. In this experiment, we compare the bandwidth usage of *partial* mode and *full* mode. We measure the average bandwidth usage of one client for both over one minute, with the cache already warmed up. Figure 3 shows that the *partial* mode uses only about 12% of bandwidth compared to the *full* mode.

F. Cache warm up

The following experiments compare both *partial* and *full* modes latencies when the cache is still cold, i.e. no objects are stored in the client side. Figure 4 shows the latency of operations during the first 10 seconds of running the application, with a cold cache. In this case, the *partial* mode produces lower latencies as it does not need to replicate the full object. The difference is more noticeable for post reading operations, as shown in Figure 4a, as the set of posts (forums) are large objects. But even for smaller objects, such as comment trees, the *partial* mode outperforms the *full* one (Figure 4b). Notice that the cache size limit does not impact these experiments, since after 10 seconds, the cache does not get full.

G. Discussion

We have seen that partial replication has advantages over full replication of objects. First, it sets an upper bound on the latency of operations by limiting the amount of data that is fetched from the store. Plus, blind update operations gain the additional benefit of being applied locally even if the object is not cached. Second, the cache is more efficiently used, which allows more objects to be kept locally even with a small cache size limit. This is useful for memory-thin devices, and to work on very large data structures with a low memory usage. Third, partial replication also reduces the bandwidth usage of the application by a factor of 8, which is especially valuable on mobile wireless connections, such as EDGE and 3G. Finally, the last advantage is a lower cost of filling the cache when starting the application. When the cache is empty all operations induce a cache miss, which is especially costly if a large object has to be fetched. Partial replication limits this issue by only replicating the parts of the object that are actually needed.

Unfortunately, *partial* mode limits the cache hit rate, as objects are not fully replicated right away, and non-replicated parts may be needed by subsequent operations. Thus, its use may depend on the workload and the cost of a cache miss. However, a tradeoff is possible between the two: instead of only fetching the parts needed by the operations, one could fetch extra parts of the object. This would however increase bandwidth and cache size utilisation. Latency could be kept low by asynchronously fetching the additional parts.

V. Related work

PRACTI [19] allows clients to select a subset of objects to replicate. Clients only receive updates on objects of their selected subset. However, clients are forced to keep some metadata about objects that they are not interested. Polyjuz [20] stores objects consisting of a set of fields. Clients can decide which fields of each object to replicate. Each subset of fields is denoted as fidelity level. Clients can select different fidelity levels according to the space or network limitations of the device where the objects are replicated. Polyjuz transparently handles the replication of an object in different fidelity levels. In Cimbiosys [21], objects are grouped into collections. Users can use filter expressions to only replicate objects that satisfy some criteria. For example, a user can group his emails in a collection and choose only to replicate emails from his university in his phone. While in the first two systems, users choose the object or fields to replicate based on their name or type, in Cimbiosys user can define replication criteria based on the value of some properties of objects.

VI. Conclusion and future work

We have introduced and formalized a new set of CRDTs called Conflict-free Partially Replicated Data Types, an extension of CRDTs which allows replicas to hold parts of data structures. Our extensive evaluation has shown that CPRDTs can improve the bandwidth and memory usage of replicas by only replicating parts needed by clients, specially in the presence of large data structures under limited cache sizes. Although cache sizes may be larger in the future, we believe that our reasoning will still apply and future applications will still benefit from the CPRDTs approach. The experimental study has also shown that CPRDTs reduce latency in average in comparison to the full mode. However, CPRDTs have a negative impact on the cache hit rate, which has to be weighted against the upper bound on the latency provided.

We plan to extend this work in several directions. First, partial replication can be used as a security mechanism to avoid replicating sensitive data by restricting access with finely grained rules. We believe it is an interesting way of exploiting CPRDTs. Second, we want to study how predictive caching techniques could still improve bandwidth usage and consequently reduce latency even more.

Acknowledgments: We thank Marek Zawirski for his help integrating CPRDTs into SwiftCloud. This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n^{o} 609551 and by the Erasmus Mundus Joint Doctorate Programme under Grant Agreement 2012-0030.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[2] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *SOSP'11*. New York, NY, USA: ACM, 2011, pp. 401–416.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[4] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and http chunking in web search," in *Velocity Web Performance and Operations Conference*, June 2009.

[5] C. Jay, M. Glencross, and R. Hubbold, "Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment," *ACM Trans. Comput.-Hum. Interact.*, vol. 14, no. 2, Aug. 2007.

[6] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: a highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, p. 447459, Apr 1990.

[7] 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.* ACM, 1995, vol. 29.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," INRIA, Rapport de recherche RR-7506, Jan. 2011.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6976, pp. 386–400.

[10] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça, "Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine," *CoRR*, vol. abs/1310.3107, 2013.

[11] "Amazon S3," http://aws.amazon.com/s3.

[12] "Google cloud storage," http://cloud.google.com/storage.

[13] "Windows Azure," http://www.microsoft.com/windowsazure.

[14] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *MobiSys'09*. Association for Computing Machinery, Inc., June 2009.

[15] "About reddit," http://www.reddit.com/about/, accessed: 2014-06-02.

[16] "Reddit source code," https://github.com/reddit/reddit, accessed: 2014-04-08.

[17] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro, "Incremental stream processing using computational conflict-free replicated data types," in *CloudDP'13*. New York, NY, USA: ACM, 2013, pp. 31–36.

[18] I. Briquemont, "Optimising client-side geo-replication with partially replicated data structures," Master's thesis, ICTEAM Institute, Universit catholique de Louvain, Sep. 2014.

[19] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "Practi replication," in *NSDI'06*. Berkeley, CA, USA: USENIX Association, 2006, pp. 5–5.

[20] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *MobiSys '09*. New York, NY, USA: ACM, 2009, pp. 83–94.

[21] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: A platform for content-based partial replication," in *NSDI'09*. Berkeley, CA, USA: USENIX Association, 2009, pp. 261–276.

D Papers under submission and technical reports

D.1 Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha and Carla Ferreira. Composition of State-based CRDTs. Technical report, 2015.

Composition of State-based CRDTs

Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha and Carla Ferreira

May 25, 2015

1 Introduction

State-based CRDTs are rooted in mathematical structures called join-semilattices (ore simply lattices, in this context). These order structures ensure that the replicated states of the defined data types evolve and increase in a partial order in a sufficiently defined way, so as to ensure that all concurrent evolutions can be merged deterministically. In order to build, or understand the building principles, of state-based CRDTs it is necessary to understand the basic building blocks of the support lattices and how lattices can be composed.

2 From Sets to Lattices

In this context the most basic structure to define is a set of distinct values. An example is the set of vowels that can defined by extension as $\mathsf{vowels} \doteq \{a, e, i, o, u\}$. Elements in a set have no specific order and they only need to be distinguishable, i.e. by defining $=$.

Having a set we can define partial orders by defining a poset over a support set and an order relation $\sqsubseteq$. This relation can be any binary relation that is reflexive, transitive and anti-symmetric. Given elements o, p, q in a set.

- (reflexive) $o \sqsubseteq o$
- (transitive) $o \sqsubseteq p \land p \sqsubseteq q \Rightarrow o \sqsubseteq q$
- (anti-symmetric) $o \sqsubseteq p \land p \sqsubseteq o \Rightarrow o = p$

Since sets already define $=$ it is possible to create posets transitively by enumerating the element pairs related by $\sqsubset$. As an example, we can build a poset with a total order on the set of vowels by defining $\langle \mathsf{vowels}, \{(a, e), (e, i), (i, o), (o, u)\}\rangle$ In this example we ordered all elements and thus created a chain, with $a \sqsubset e \sqsubset i \sqsubset o \sqsubset u$, i.e. given any two elements o, p either $o \sqsubseteq p$ or $p \sqsubseteq o$.

If some elements were left unordered we could have concurrent elements.

- (concurrent) $o \parallel p \iff \neg(o \sqsubseteq p \lor p \sqsubseteq o)$

In the extreme case we could have left all elements unordered and defined a poset that depicted an *antichain*, where any two elements are always concurrent. E.g. $\langle \mathsf{vowels}, \{\}\rangle$. Having a poset we also have the properties of a set.

$$\frac{A : \mathsf{poset}}{A : \mathsf{set}}$$

For a given poset to be a join-semilattice there must be a *least-upper-bound* for any subset of the support set. Given a pair of elements o, p, their least-upper-bound can be derived by the result of a binary join operator, by $o \sqcup p$. Since the binary join is commutative and associative it can be iterated over the elements of any subset to derive the least-upper-bound of the subset. Some properties of join are listed bellow.

- (idempotent) $o \sqcup o = o$
- (commutative) $o \sqcup p = p \sqcup o$
- (associative) $o \sqcup (p \sqcup q) = (o \sqcup p) \sqcup q$

And properties of least-upper-bounds.

- (upper-bound) $o \sqsubseteq o \sqcup p$
- (least-upper-bound) $o \sqsubseteq q \wedge o \sqsubseteq q \Rightarrow o \sqsubseteq p \sqcup q$

A general example of a poset with a join is obtained from any set by selecting the order to be set inclusion and the join to be set union. In our running example this would be the $\mathsf{lattice}$ defined by $\langle \mathsf{vowels}, \subseteq, \cup\rangle$. Another simple lattice can be obtained by taking the maximum in a total order (or dually the minimum), for naturals we can derive $\mathsf{maxint} \doteq \langle \mathbb{N}, \leq_{\mathbb{N}}, \mathbf{max}\rangle$.

Having a $\mathsf{lattice}$ we also have the properties of a poset.

$$\frac{A : \mathsf{lattice}}{A : \mathsf{poset}}$$

A chain (a special case of a poset) always derives a $\mathsf{lattice}$.

$$\frac{A : \mathsf{chain}}{A : \mathsf{lattice}}$$

Notice that although some specific partial orders always derive lattices, as is the case for *chains*, in general we can have partial orders that are not lattices. An example is the prefix ordering on bit strings that can produce concurrent elements, $010 \parallel 100$, and is not a lattice.

We will see in latter sections that in some cases it is useful to have a special element in the lattice that is the bottom element $\bot$. Some properties are.

- (bottom) $\bot \sqsubseteq o$
- (identity) $\bot \sqcup o = o$

The lattice formed by set inclusion has the empty set as bottom, $\langle \mathsf{vowels}, \subseteq, \cup, \emptyset \rangle$. Not all lattices have a "natural" bottom, but it is always possible to add an extra element as bottom to an existing lattice. We will address this construction when talking about lattice composition by linear sums. As expected, lattices with bottom also have the lattice properties.

$$\frac{A : \mathsf{lattice}_\perp}{A : \mathsf{lattice}}$$

2.1 Primitive Lattices

We now introduce a small set of lattices, that will be later useful to construct more complex structures by composition.

Singleton A single element, $\perp$.

$$\frac{}{\mathbb{1} : \mathsf{lattice}_\perp}$$

$$\perp \sqsubseteq \perp \qquad \perp \sqcup \perp = \perp$$

Boolean Two elements $\mathbb{B} = \{\mathsf{False}, \mathsf{True}\}$ in a chain, join is logical $\vee$.

$$\frac{}{\mathbb{B} : \mathsf{lattice}_\perp}$$

$$\mathsf{False} \sqsubseteq \mathsf{True} \qquad x \sqcup y = x \vee y \qquad \perp = \mathsf{False}$$

Naturals Natural numbers. We include the 0, thus $\mathbb{N} = \{0, 1, \ldots\}$.

$$\frac{}{\mathbb{N} : \mathsf{lattice}_\perp}$$

$$n \sqsubseteq m = n \leq m \qquad n \sqcup m = \max(n, m) \qquad \perp = 0$$

3 Inflations make CRDTs

State-based CRDTs can be specified by selecting a given lattice to model the state, and choosing an initial value in the lattice, usually the $\perp$. Mutation operations can only change the state by *inflations* and do not return values. Query operations evaluate an arbitrary function on the state and return a value.

An inflation is an endo-function on the lattice type that picks a value x among the set of valid lattice states a and produces a new value state such that:

- (inflation) $x \sqsubseteq f(x)$

Inflations can be further classified as non-strict and strict inflations, where a strict inflation is such that:

- (strict inflation) $x \sqsubset f(x)$

We can now classify inflations.

$$\frac{\forall x \in a \cdot x \sqsubseteq f(x)}{f : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{\forall x \in a \cdot x \sqsubset f(x)}{f : A \xrightarrow{\sqsubset} A}$$

$$\frac{f : A \xrightarrow{\sqsubset} A}{f : A \xrightarrow{\sqsubseteq} A}$$

A state that is only updated as a result of an inflation over its current value, is immutable under joins with copies of past states.

Notice that an inflation is not the same as a monotonic function, $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. Example, the function $f(x) = \frac{x}{2}$ on positive reals is monotonic and is not an inflation.

3.1 Primitive Inflations

Building on the primitive lattices introduced above we can build some inflations.

$$\mathsf{id}(x) = x \qquad \frac{}{\mathsf{id} : A \xrightarrow{\sqsubseteq} A}$$

$$\underline{\mathsf{True}}(x) = \mathsf{True} \qquad \frac{}{\underline{\mathsf{True}} : \mathbb{B} \xrightarrow{\sqsubseteq} \mathbb{B}}$$

$$\mathsf{succ}(x) = x + 1 \qquad \frac{}{\mathsf{succ} : \mathbb{N} \xrightarrow{\sqsubset} \mathbb{N}}$$

3.2 Sequential Composition

Inflations can be composed sequentially. As long as there is at least one strict inflation in the composition, we are sure to also have a strict composition.

$$(f \bullet g)(x) = f(g(x))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubseteq} A}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : A \xrightarrow{\sqsubset} A}{f \bullet g : A \xrightarrow{\sqsubset} A} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : A \xrightarrow{\sqsubseteq} A}{f \bullet g : A \xrightarrow{\sqsubset} A}$$

4 Lattice Compositions

Since we are interested in creating lattices we consider a few composition techniques that are known to derive lattices. While in some cases they build from other lattices, in others they can derive lattices from simpler structures.

4.1 Product

The product $\times$, or pair construction, derives a lattice formed by pairs of other lattices. It can be applied recursively and derive a composition from a sequence of lattices, where operations are applied in point-wise order.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \times B : \mathsf{lattice}}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$$

$$(x_1, y_1) \sqcup (x_2, y_2) = (x_1 \sqcup x_2, y_1 \sqcup y_2)$$

The construction also extends to $\mathsf{lattice}_\perp$ when all sources are also $\mathsf{lattice}_\perp$.

$$\frac{A : \mathsf{lattice}_\perp \qquad B : \mathsf{lattice}_\perp}{A \times B : \mathsf{lattice}_\perp}$$

$$\perp = (\perp, \perp)$$

As an example, the underlying lattice structure of a version vector among three replica nodes is composable by $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ with $\perp = (0, 0, 0)$.

Bellow are the properties of inflations over products. A strict inflation on one of the components leads to an overall strict inflation.

$$(f \times g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubseteq} A \times B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \times g : A \times B \xrightarrow{\sqsubset} A \times B}$$

4.2 Lexicographic Product

The $\boxtimes$ construct builds a lexicographic order from its source lattices. Components to the left are more significant and, unless they are equal, they filter out further comparisons to the right side.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}_\perp}{A \boxtimes B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\perp \qquad B : \mathsf{lattice}_\perp}{A \boxtimes B : \mathsf{lattice}_\perp}$$

$$(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubseteq x_2 \vee (x_1 = x_2 \wedge y_1 \sqsubseteq y_2)$$

$$(x_1, y_1) \sqcup (x_2, y_2) = \begin{cases} (x_1, y_1) & \textbf{if } x_2 \sqsubset x_1 \\ (x_2, y_2) & \textbf{if } x_1 \sqsubset x_2 \\ (x_1, y_1 \sqcup y_2) & \textbf{if } x_1 = x_2 \\ (x_1 \sqcup x_2, \bot) & \textbf{otherwise} \end{cases}$$

$$\bot = (\bot, \bot)$$

In the join definition we can observe that the $\bot$ value is used only when the left components can have concurrent values. Note that B could be simply a lattice (B : $\mathsf{lattice}$) and then join definition could be redefined in the following manner:

$$(x_1, y_1) \sqcup (x_2, y_2) = \begin{cases} (x_1, y_1) & \textbf{if } x_2 \sqsubset x_1 \\ (x_2, y_2) & \textbf{if } x_1 \sqsubset x_2 \\ (x_1, y_1 \sqcup y_2) & \textbf{if } x_1 = x_2 \\ (x_1 \sqcup x_2, y_1 \sqcup y_2) & \textbf{otherwise} \end{cases}$$

If the left component is a chain, often the case in practical uses, then the right one can be a simple $\mathsf{lattice}$ (without $\bot$) and the fourth clause of the join definition is not used.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{lattice}}{A \boxtimes B : \mathsf{lattice}}$$

And, if the right component is also a chain the composition is a chain.

$$\frac{A : \mathsf{chain} \qquad B : \mathsf{chain}}{A \boxtimes B : \mathsf{chain}}$$

Properties of inflations.

$$(f \boxtimes g)(x, y) = (f(x), g(y))$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubseteq} A \boxtimes B}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubset} B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B} \qquad \frac{f : A \xrightarrow{\sqsubset} A \quad g : B \longrightarrow B}{f \boxtimes g : A \boxtimes B \xrightarrow{\sqsubset} A \boxtimes B}$$

Notice that if we apply a strict inflation to the left component, then the right can be transformed by any (endo-)function even if non inflationary. In practice this allows resetting the right component after strictly inflating the left.

4.3 Linear Sum

The next composition, linear sum $\oplus$, picks two lattices, left and right, and creates a new lattice where any element from the left lattice is always lower that any element in the right lattice. In the resulting set the elements are tagged with a label that identifies from which source lattice they came form. i.e. $\mathsf{Left}\ a$ means that element a came from the left lattice and is now named $\mathsf{Left}\ a$. Tagging also ensures that the sets supporting each lattice could have had elements in common.

$$\frac{A : \mathsf{lattice} \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}} \qquad \frac{A : \mathsf{lattice}_\perp \qquad B : \mathsf{lattice}}{A \oplus B : \mathsf{lattice}_\perp}$$

$$\begin{aligned}
\mathsf{Left}\ x &\sqsubseteq \mathsf{Left}\ y = x \sqsubseteq y & \mathsf{Left}\ x &\sqcup \mathsf{Left}\ y = \mathsf{Left}\ (x \sqcup y)\\
\mathsf{Right}\ x &\sqsubseteq \mathsf{Right}\ y = x \sqsubseteq y & \mathsf{Right}\ x &\sqcup \mathsf{Right}\ y = \mathsf{Right}\ (x \sqcup y)\\
\mathsf{Left}\ x &\sqsubseteq \mathsf{Right}\ y = \mathsf{True} & \mathsf{Left}\ x &\sqcup \mathsf{Right}\ y = \mathsf{Right}\ y\\
\mathsf{Right}\ x &\sqsubseteq \mathsf{Left}\ y = \mathsf{False} & \mathsf{Right}\ x &\sqcup \mathsf{Left}\ y = \mathsf{Right}\ x
\end{aligned}$$

$$\perp = \mathsf{Left}\ \perp$$

A possible use of this construction is to add a $\perp$ to a lattice that didn't had one. For instance $\mathbb{1} \oplus \mathbb{R}$ can add a special element, e.g. nil, that is ordered as lower than any real number. The same construction can also be used to add a top element $\top$ to a lattice, that can act as a tombstone that stops lattice evolution. Notice that for any state x, $x \sqcup \top = \top$.

Properties of inflations.

$$\begin{aligned}
(f \oplus g)(\mathsf{Left}\ x) &= \mathsf{Left}\ f(x)\\
(f \oplus g)(\mathsf{Right}\ x) &= \mathsf{Right}\ g(x)
\end{aligned}$$

$$\frac{f : A \xrightarrow{\sqsubseteq} A \quad g : B \xrightarrow{\sqsubseteq} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubseteq} A \oplus B}$$

$$\frac{f : A \xrightarrow{\sqsubset} A \quad g : B \xrightarrow{\sqsubset} B}{f \oplus g : A \oplus B \xrightarrow{\sqsubset} A \oplus B}$$

4.4 Function and Map

A total function $\rightarrow$ is obtained by combining a set with a $\mathsf{lattice}$. This construction does keywise comparison and joins.

$$\frac{A : \mathsf{set} \qquad B : \mathsf{lattice}}{A \rightarrow B : \mathsf{lattice}} \qquad \frac{A : \mathsf{set} \qquad B : \mathsf{lattice}_\perp}{A \rightarrow B : \mathsf{lattice}_\perp}$$

$$f \sqsubseteq g = \forall x \in A \cdot f(x) \sqsubseteq g(x) \qquad (f \sqcup g)(x) = f(x) \sqcup g(x)$$

$$\perp(x) = \perp$$

A map $\hookrightarrow$ can be obtained from a function by assigning a bottom to keys that are not present in a given map, and then using the function definitions. The linear sum construction is used to assign a distinguished bottom to any $\mathsf{lattice}$ V in the co-domain.

$$K \hookrightarrow V \cong K \to \mathbb{1} \oplus V$$

$$\frac{K : \mathsf{set} \qquad V : \mathsf{lattice}}{K \hookrightarrow V : \mathsf{lattice}_\perp}$$

For example, we can define a map of vowels keys to integer counters $\mathsf{vowels} \hookrightarrow \mathbb{N}$ by using a total function $\mathsf{vowels} \to \mathbb{1} \oplus \mathbb{N}$. Where the map state $\{a \mapsto 3, i \mapsto 5\}$ would be the same as the function state $\{a \mapsto 3, e \mapsto \perp, i \mapsto 5, o \mapsto \perp, u \mapsto \perp\}$.

We define some inflations over maps. The first inflation applies an inflation to all values in the co-domain and thus inflates the map composition.

$$\mathsf{map}(f)(m) = \{(k, f(v)) \mid (k, v) \in m\}$$

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{map}(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

The second inflation transforms the value on a given key, and if the key is missing applies it to $\perp$. This allows a strict inflation in the co-domain lattice to imply a strict inflation in the composition.

$$\mathsf{apply}_k(f)(m) = \begin{cases} m\{k \mapsto f(v)\} & \textbf{if } (k, v) \in m \\ m\{k \mapsto f(\perp)\} & \textbf{otherwise} \end{cases}$$

$$\frac{f : V \xrightarrow{\sqsubseteq} V}{\mathsf{apply}_k(f) : (K \hookrightarrow V) \xrightarrow{\sqsubseteq} (K \hookrightarrow V)}$$

$$\frac{f : V \xrightarrow{\sqsubset} V}{\mathsf{apply}_k(f) : (K \hookrightarrow V) \xrightarrow{\sqsubset} (K \hookrightarrow V)}$$

4.5 Sets and Multisets

Given any set A it is possible to derive a $\mathsf{lattice}_\perp$ by using the set of all possible subsets, the *powerset* $\mathcal{P}(A)$.

For example, $\mathcal{P}(\{x, y, z\}) = \{\{\}, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$.

$$\frac{A : \mathsf{set}}{\mathcal{P}(A) : \mathsf{lattice}_\perp}$$

$$\mathcal{P}(A) \cong A \to \mathbb{B}$$

$$a \sqsubseteq b = a \subseteq b \qquad a \sqcup b = a \cup b \qquad \bot = \{\}$$

The *powerset* can also be represented by a function composition that maps each set element to a boolean that states its presence in the subset.

This composition is very general since it can produce a $\mathsf{lattice}_\bot$ from any set.

A natural extension is to represent *mutisets* by mapping the domain set to naturals, instead of booleans.

$$\frac{A : \mathsf{set}}{\mathcal{M}(A) : \mathsf{lattice}_\bot}$$

$$\mathcal{M}(A) \cong A \to \mathbb{N}$$

$$a \sqsubseteq b = a \subseteq b \qquad a \sqcup b = a \cup b \qquad \bot = \{\}$$

The generic inflations defined for functions when used here show that adding elements is inflationary. For sets represented by $A \to \mathbb{B}$ with a given state s we can define how to add an element e.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\underline{\mathsf{True}})(s)$$

Likewise, when adding on multisets $A \to \mathbb{N}$ one increments the element count, having a strict inflation.

$$\mathsf{add}(e)(s) = \mathsf{apply}_e(\mathsf{succ})(s)$$

4.6 Antichain of Maximal Elements

Starting from a poset this construction produces a $\mathsf{lattice}_\bot$ by keeping an antichain of maximal elements, given the base poset order. Upon join, all elements that are concurrent are kept, but any element that is present together with a higher element is removed.

$$\frac{A : \mathsf{poset}}{\mathcal{A}(A) : \mathsf{lattice}_\bot}$$

$$\mathcal{A}(A) = \{\mathsf{maximal}(a) \mid a \in \mathcal{P}(A)\}$$

$$\mathsf{maximal}(a) = \{x \in a \mid \nexists y \in a \cdot x \sqsubset y\}$$

$$a \sqsubseteq b = \forall x \in a \cdot \exists y \in b \cdot x \sqsubseteq y$$

$$a \sqcup b = \mathsf{maximal}(a \cup b)$$

$$\bot = \{\}$$

5 Abridged Catalog

In order to exemplify the composition constructs we present a small set of example CRDTs. Simple query functions are included and all mutators are inflations.

Notice that join does not need to be defined as it follows from the composition rules that were introduced.

5.1 Positive Counter

This simple form of counter can only increase. Replica nodes must have access to unique ids among a set I and can only increment its position in a map of ids to integers. While increment mutators are parametrized by id i the query is anonymous and simply inspects the state.

$$\mathsf{PCounter} = I \hookrightarrow \mathbb{N}$$

$$\begin{aligned}\mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(a)\\ \mathsf{value}(a) &= \sum\{v \mid (i,v) \in a\}\end{aligned}$$

Notice that if a given node does not yet have an entry in the map and increments, then succ applies over $\bot$, which for $\mathbb{N}$ was defined to be 0.

Positive counter with reset

$$\mathsf{PCounter} = (I \hookrightarrow \mathbb{N}) \times (I \hookrightarrow \mathbb{N})$$

$$\begin{aligned}\mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(\mathsf{fst}(a))\\ \mathsf{reset}(a) &= \bot \times \mathsf{fst}(a) \sqcup \mathsf{snd}(a)\\ \mathsf{value}(a) &= \sum\{v \mid (i,v) \in \mathsf{fst}(a)\} - \sum\{v \mid (i,v) \in \mathsf{snd}(a)\}\end{aligned}$$

5.2 Positive and Negative Counter

This variation allows for both increments and decrements. A solution is to pair two positive counters and consider the right side as negative. We use the standard functions fst() and snd() to respectively access the left and right elements of a pair.

$$\mathsf{PNCounter} = I \hookrightarrow \mathbb{N} \times I \hookrightarrow \mathbb{N}$$

$$\begin{aligned}\mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(\mathsf{fst}(a))\\ \mathsf{dec}_i(a) &= \mathsf{apply}_i(\mathsf{succ})(\mathsf{snd}(a))\\ \mathsf{value}(a) &= \sum\{v \mid (i,v) \in \mathsf{fst}(a)\} - \sum\{v \mid (i,v) \in \mathsf{snd}(a)\}\end{aligned}$$

An alternative way to obtain a similar result is to use a lexicographic pair and have the first element incremented when one needs to update the count on the second element.

$$\mathsf{PNCCounter} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{Z}$$

$$\begin{aligned}
\mathsf{inc}_i(a) &= \mathsf{apply}_i(\mathsf{id} \boxtimes \mathsf{succ})(a) \\
\mathsf{dec}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \mathsf{pred})(a) \\
\mathsf{value}(a) &= \sum\{\mathsf{snd}(v) \mid (i, v) \in a\}
\end{aligned}$$

$$\mathsf{pred}(x) = x - 1$$

5.3 Observed-remove Add-wins Set

An observed-remove set with add-wins semantics can be derived by creating unique tokens whenever a new element is inserted, using for that a grow only counter per replica, and canceling this tokens, by increasing a boolean to True, upon removal. Only elements supported by non-canceled tokens are considered to be in the set.

$$\mathsf{ORSet}^{+} = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$\begin{aligned}
\mathsf{add}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\
\mathsf{rmv}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a)
\end{aligned}$$

$$\mathsf{member}_e(a) = \exists (e, m) \in a \cdot \exists i, n \cdot (n, \mathsf{False}) \in m(i)$$

5.4 Observed-remove Remove-wins Set

An observed-remove set with remove-wins semantics is derived by a dual construction to the previous one, while sharing the same state lattice. Removal creates unique tokens, and additions need to cancel all remove tokens that are visible in the state.

$$\mathsf{ORSet}^{-} = E \hookrightarrow I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$
\begin{aligned}
\mathsf{rmv}_{e,i}(a) &= \mathsf{apply}_e(\mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}}))(a) \\
\mathsf{add}_e(a) &= \mathsf{apply}_e(\mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}}))(a) \\
\\
\mathsf{member}_e(a) &= \exists (e,m) \in a \cdot \nexists i, n \cdot (n, \mathsf{False}) \in m(i)
\end{aligned}
$$

5.5 Enable-wins Flag

A boolean flag that can be flipped, implemented in Riak under the name flag data type. It is a special case of an ORSet^{+} for a singleton set. Flag starts disabled.

$$\mathsf{Flag}^{+} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$
\begin{aligned}
\mathsf{enable}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}})(a) \\
\mathsf{disable}(a) &= \mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}})(a) \\
\\
\mathsf{value}(a) &= \exists i, n \cdot (n, \mathsf{False}) \in a(i)
\end{aligned}
$$

5.6 Disable-wins Flag

A boolean flag that can be flipped, implemented in Riak under the name flag data type. It is a special case of an ORSet^{-} for a singleton set. Flag starts enabled.

$$\mathsf{Flag}^{-} = I \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}$$

$$
\begin{aligned}
\mathsf{disable}_i(a) &= \mathsf{apply}_i(\mathsf{succ} \boxtimes \underline{\mathsf{False}})(a) \\
\mathsf{enable}(a) &= \mathsf{map}(\mathsf{id} \boxtimes \underline{\mathsf{True}})(a) \\
\\
\mathsf{value}(a) &= \nexists\, i, n \cdot (n, \mathsf{False}) \in a(i)
\end{aligned}
$$

5.7 Multi-value Register

A non-optimized multi-value register can be derived by lexicographic coupling of a version vector clock C with a payload value V. When a new value v is to be assigned, a new clock, greater than all visible clocks in the state, is created and coupled with the value. These pairs are kept in a antichain of maximal elements. Thus, upon merge, concurrently assigned values will be collected, but any subsequent assignment will again reduce the state to a single pair value.

$$\begin{aligned}\mathsf{MVReg}(V, I) &= \mathcal{A}(C \boxtimes V)\\ C &= I \hookrightarrow \mathbb{N}\end{aligned}$$

$$\begin{aligned}\mathsf{assign}_{v,i}(a) &= \{\mathsf{apply}_i(\mathsf{succ})(\bigsqcup\{c \mid (c, v') \in a\}) \boxtimes v\}\\ \mathsf{values}(a) &= \{v \mid (c, v) \in a\}\end{aligned}$$

Notice that the value is never updated without creating a new clock. Thus, lexicographic comparison (needed for the operation of the antichain join) is always decided by the first component, and in practice V can be any opaque payload without need to define a partial order on its values.

6 Closing Remarks

This report collects several composition techniques for lattices, adopts the notion of inflation and how it applies to the specification of state based CRDTs over lattices. Most of the lattice compositions are very standard techniques from order theory [5]. An early version of this work was presented at Schloss Dagstuhl under the title *Composition of Lattices and CRDTs* and the summary of the presentation is available at [6]. Most of the CRDT constructions used here are influenced by work in [8, 7, 2, 4, 3, 1].

References

[1] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Proceedings of Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference.* Springer, 2014.

[2] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In Marcos K. Aguilera, editor, *Int. Symp. on Dist. Comp. (DISC)*, volume 7611 of *Lecture Notes in Comp. Sc.*, pages 441–442, Salvador, Bahia, Brazil, October 2012. Springer-Verlag.

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

Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 271–284. ACM, 2014.

[4] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[5] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[6] Bettina Kemme, André Schiper, G. Ramalingam, and Marc Shapiro. Dagstuhl seminar review: Consistency in distributed systems. *SIGACT News*, 45(1):67–89, March 2014.

[7] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011.

[8] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag.

D.2 Carlos Baquero, Nuno Preguiça. Why logical clocks are easy. Submitted to ACM Queue, 2015.

Why logical clocks are easy

Carlos Baquero, Nuno Preguiça

July 26, 2015

We can say that any computing system executes sequences of actions, with an action being any relevant change in the state of the system. For example, reading a file to memory, modifying the contents of the file in memory, or writing the new contents to the file, are relevant actions for a text editor. In a distributed system, actions execute in multiple locations; in this context, actions are often named events. Examples of events in distributed systems include sending or receiving messages, or changing some state in a node. Not all events are related, but some events can cause and influence how other, latter events, occur. For instance a reply to a received mail message is influenced by that message, and maybe by other prior messages also received.

Events in a distributed system can either occur in a close location such as different processes running in the same machine, at nodes inside a data center, or geographically spread across the globe, or even at a larger scale in the near future. Relations of potential cause and effect between events are fundamental to the design of distributed algorithms, and nowadays few services can claim not to have some form of distributed algorithm at its core.

Before we try to make sense of these cause and effect relations, it is necessary to limit their scope to what can be perceived inside the distributed system itself – we can refer to this as *internal causality.* Naturally, a distributed system interacts with the rest of the physical world outside it, and there are also cause and effect relations in that world at large. For example, consider a couple planning a night out using a system that manages reservations for dinners and cinema. One person reserves the dinner and calls the other on the phone saying that. After receiving the phone call, the second person goes to the system and reserves the cinema. In a distributed system, the system has no way to know that the first reservation has actually caused the second one.

This *external causality* cannot be detected by the system, and can only be approximated by *physical time* (however, time totally orders all events, even those unrelated, thus it is no substitute to causality; and wall clocks are never perfectly synchronized [14]). In this article, we focus on characterizing *internal causality*, the causality that can be tracked by the system.

Happened-before relation This brings us to 1978, when Leslie Lamport defined a partial order, *happened before*, that connects events of a distributed systems that are potentially causally linked [7]. An event c can be the cause of an event e, or c happened before e, iff both occur in the same process and c executed first, or, being at different processes, if e could know the occurrence of c thanks to some message received from some process that knows about c. If neither event can know about the other, we say they are concurrent.

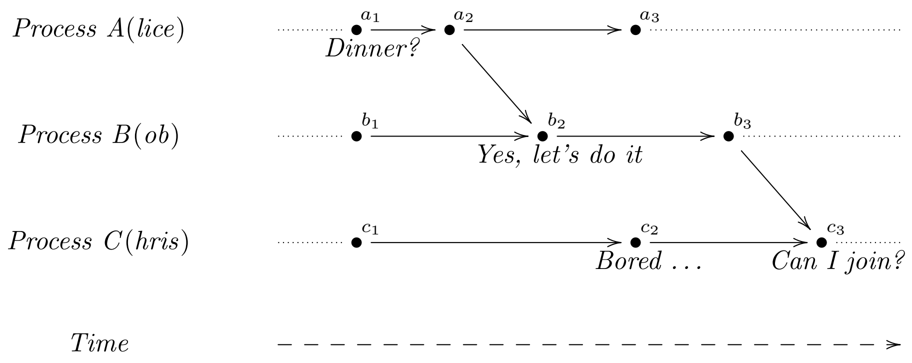


Figure 1: Run in a distributed system with three nodes: happens-before relation.

Figure 1 shows an example of a distributed system. An arrow between processes represents a message sent and delivered. We can see that both Bob's positive answer to the dinner suggestion by Alice, and Chris later request to join the party, are both influenced by Alice's initial question about plans for dinner.

Looking at the events in this distributed computation, a simple way to check if an event c could have caused another event e (c happened before e) is to find at least one directed path linking c to e. If such a connection is found we mark this partial order relation by $c \to e$ to denote the happened before relation or potential causality. For instance we have $a_1 \to b_2$ and $b_2 \to c_3$ (and yes, as well $a_1 \to c_3$, since causality is transitive). Events a_1 and c_2 are concurrent, denoted $a_1 \parallel c_2$, because there are no causal paths in either direction. We note $x \parallel y$ iff $x \not\to y$ and $y \not\to x$. The fact that Chris was bored didn't influence Alice's question about dinner, nor the other way around.

We can now recapitulate the three possible relations between two events x and y: (a) x might have influenced y, if $x \to y$; (b) y might have influenced x, if $y \to x$; (c) no known influence among x and y, as they occurred concurrently $x \parallel y$.

Causal histories Causality can be tracked in a very simple way by using *causal histories* [12, 2]. The system can locally assign unique names to each event (e.g. process name and local increasing counter) and collect, and transmit, sets of events to capture the known past.

For a new event, the system creates a new unique name and the causal history is comprised of the union of this name and the causal history of the previous event in the process. For example, the second event in process C is

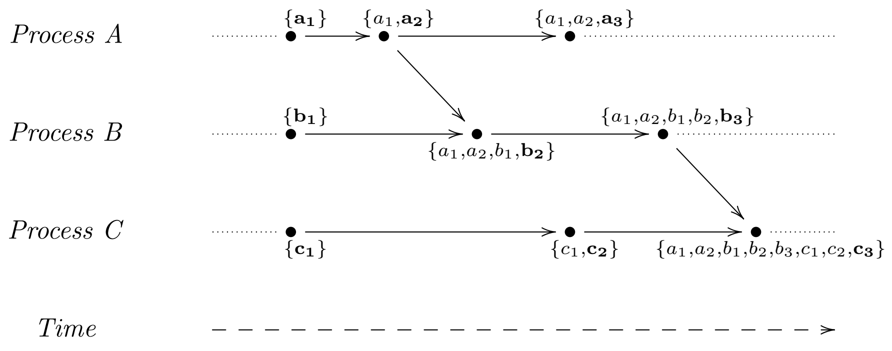

Process A
Process B
Process C
Time
$\{\mathbf{a_1}\}$
$\{a_1,\mathbf{a_2}\}$
$\{a_1,a_2,\mathbf{a_3}\}$
$\{\mathbf{b_1}\}$
$\{a_1,a_2,b_1,\mathbf{b_2}\}$
$\{a_1,a_2,b_1,b_2,\mathbf{b_3}\}$
$\{\mathbf{c_1}\}$
$\{c_1,\mathbf{c_2}\}$
$\{a_1,a_2,b_1,b_2,b_3,c_1,c_2,\mathbf{c_3}\}$

Figure 2: Run in a distributed system with three nodes: causal histories.

assigned name c_2 and its causal history is $H_c = \{c_1, \mathbf{c_2}\}$ (shown in Figure 2). When a process sends a message, the causal history of the send event is sent with the message. On reception, the remote causal causal history is merged (by set union) to the local history. For example, the delivery of the first message from process A to B merges the remote causal history, $\{a_1, a_2\}$, with the local history, $\{b_1\}$, and the new unique name, b_2, leading to $\{a_1, a_2, b_1, \mathbf{b_2}\}$.

Checking causality between two events x and y, can be tested simply by set inclusion: $x \to y$ iff $H_x \subset H_y$. This follows from the definition of causal histories, where the causal history of an event will be included in the causal history of the following event. Even better, if we mark the last local event added to the history (distinguished in **bold** in the diagram) we can use a simpler test: $x \to y$ iff $x \in H_y$ – e.g. $a_1 \to b_2$, since $a_1 \in \{a_1, a_2, b_1, b_2\}$. This follows from the fact that a causal history includes all events that (causally) precede a given event.

Vector clock It should be obvious by now, that causal histories work but are not very compact. For instance, the mechanism of building the history implies that if an event b_3 is present in H_y, then all preceding events from that same process, b_1 and b_2, are also present in H_y. Thus, its suffices to store the most recent event from each process. Causal history $\{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3\}$ is compacted to $\{a \mapsto 2, b \mapsto 3, c \mapsto 3\}$, or simply a vector $[2, 3, 3]$.

Now, we can translate the rules used with causal histories to the new compact vector representation.

For verifying that $x \to y$, we needed to check if $H_x \subset H_y$. This can be done, verifying for each node, if the unique names contained in H_x are also contained in H_y and there is at least one unique name in H_y that is not contained in H_x. This is immediately translated in checking if each entry in the vector of x is smaller or equal to the correspondent entry in the vector of y and one is strictly smaller, i.e., $\forall i : V_x[i] \leq V_y[i]$ and $\exists j : V_x[j] < V_y[j]$. This can be stated more compactly by $x \to y$ iff $V_x < V_y$.

For a new event, the creation of a new unique name is equivalent to incre-

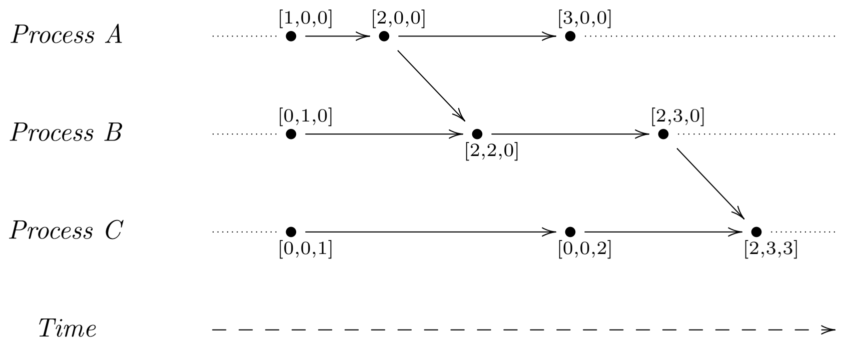


Figure 3: Run in a distributed system with three nodes: vector clocks.

menting the entry in the vector for the process where the event is created. For example, the second event in process c has vector $[0, 0, 2]$, that corresponds to the creation of event c_2 of the causal history.

Finally, doing the union of two causal histories, H_x and H_y, is equivalent to taking the point-wise maximum of the correspondent two vectors V_x and V_y, i.e., $\forall i : V[i] = \mathbf{max}(V_r[i], V_l[i])$. The intuition is that, for the unique names generated in each node, we only need to keep the one with the largest counter.

When receiving a message, besides merging the causal histories, a new event is created. The vector representation of these steps can be seen, for example when the first message from a is received in b, where taking the point-wise maximum leads to $[2, 1, 0]$ and the new unique name finally leads to $[2, 2, 0]$.

This compact representation, is known as *vector clock* and was introduced around 1988 [4, 9]. As explained, vector comparison is an immediate translation of set inclusion of causal histories. This equivalence is often forgotten in modern descriptions of vector clocks, and can make what is a simple encoding problem into an unnecessarily complex and arcane set of rules, breaking the intuition.

Dotted Vector Clocks When using causal histories, we have shown that knowing the last event could simplify comparison by simply checking if the last event is included in the causal history. This can still be done with vectors, if we keep track in which node the last event has been created. For example, when questioning if $x = [\mathbf{2}, 0, 0] \to y = [2, \mathbf{3}, 0]$, with boldface indicating the last event in each vector, we can simply test if $x[0] \leq y[0]$ ($\mathbf{2} \leq 2$) since we have marked that the last event in x was created in node a, i.e., it corresponds to the first entry of the vector. Since marking numbers in **bold** is not a very practical implementation, the last event is usually stored outside the vector (and sometimes called a *dot*): e.g. $[2, \mathbf{2}, 0]$ can be represented as $[2, 1, 0]b_2$. Notice that now the vector represents the causal past of b_2, excluding the event itself.

Version Vector In an important class of applications there is no need to register causality for all the events in a distributed computation. For instance, when modifying replicas of data, it often suffices to only register events that change replicas. In this case, when thinking about causal histories, we only need to assign a new unique name to these relevant events. Still, we need to propagate the causal histories when messages are propagated from one site to the other and the remaining rules for comparing causal histories remain unchanged.

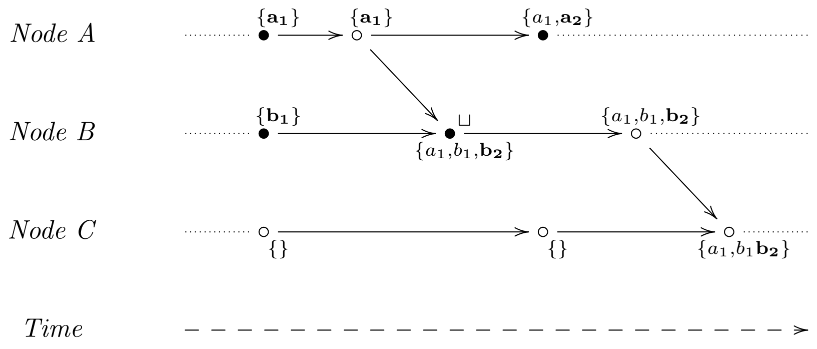


Figure 4: Run in a distributed system with three nodes, where only some events are relevant: causal histories.

Figure 4 presents the same example as before, but with events not being registered for causality tracking denoted with ◦. If the run represents the updates to replicas of a data object, we can see that after node a and b are concurrently modified, the state of replica a is sent to replica b (in a message). When the message is received in node b, it is detected that two concurrent updates have occurred, with histories $\{a_1\}$ and $\{b_1\}$, as neither $a_1 \rightarrow b_1$ nor $b_1 \rightarrow a_1$. In this case, a new version that merges the two updates is created (merge is denoted by the join symbol $\sqcup$), which requires creating a new unique name, leading to $\{a_1, b_1, b_2\}$. When the state of replica b is later propagated to replica c, as no concurrent update exist in replica c, no new version is created.

Again we can use vectors to compact the representation. The resulting representation is known as version vector and was created in 1983 [10], five years before vector clocks. Figure 5 presents the same example as before, represented with version vectors.

In some cases, when the state of one replica is propagated to the other replica, the two versions are kept by the system as conflicting versions. For example, in Figure 6, when the message from node a is received in node b, the system keeps each causal history, $\{a_1\}$ and $\{b_1\}$, associated to the respective version. The causal history associated with the node containing both version is $\{a_1, b_1\}$, the union of the causal history of all versions This approach allows to later check for causality relations between each version and other versions when merging the state of additional nodes. The conflicting versions could also be

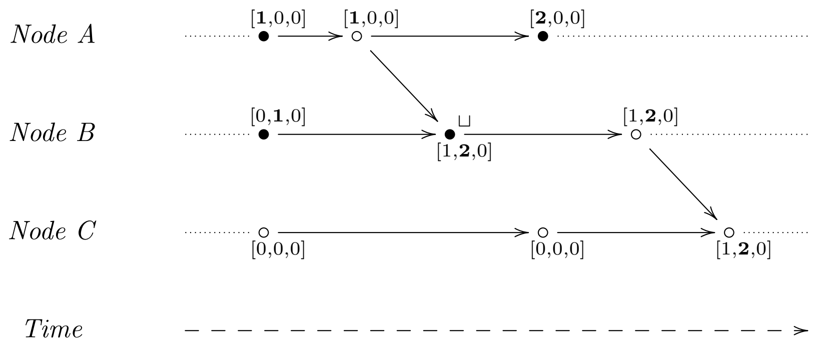


Figure 5: Run in a distributed system with three nodes, where only some events are relevant: version vectors.

merged, creating a new unique name, as in the example.

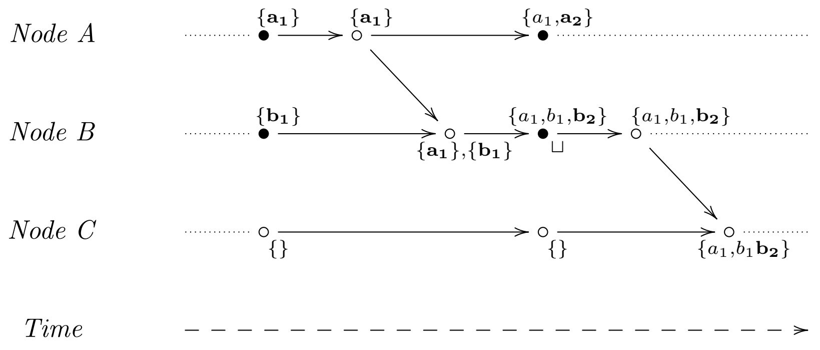


Figure 6: Run in a distributed system with three nodes, where only some events are relevant and versions are not immediately merged: causal histories.

Dotted Version Vector One limitation of causality tracking by vectors is that one entry is needed for each source of concurrency [3]. One can expect a difference of several orders of magnitude from the number of nodes in a datacenter to the number of clients they handle. Vectors with one entry per client, don't scale well when millions of clients are accessing the service [6]. Again, we can appeal to the foundation of causal histories to check how to overcome this limitation.

The basic requirement in causal histories is that each event is assigned a unique identifier. There is no requirement that this unique identifier is created locally nor that it is immediately created. Thus, in systems where nodes can be divided in clients and servers and where clients communicate only with servers, it is possible to delay the creation of a new unique name until the client communicates with the server and to use a unique name generated in the server. The causal history associated with the new version is the union of the causal history of the client and the newly assigned unique name.

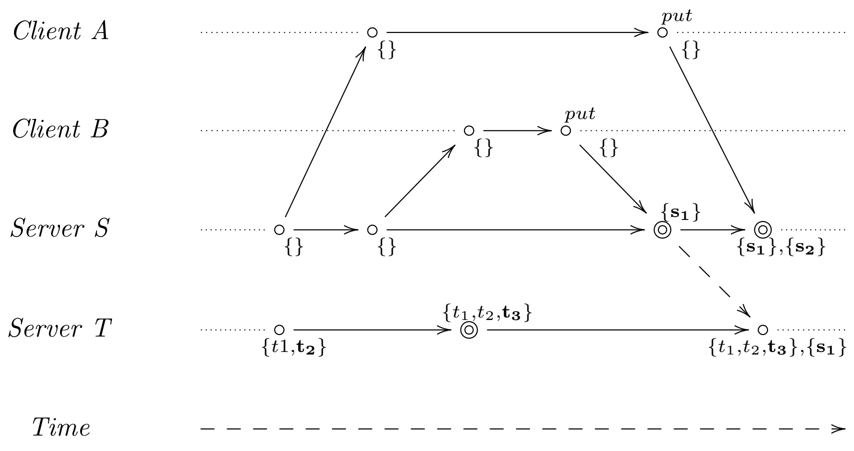


Figure 7: Run in a distributed storage system: causal histories.

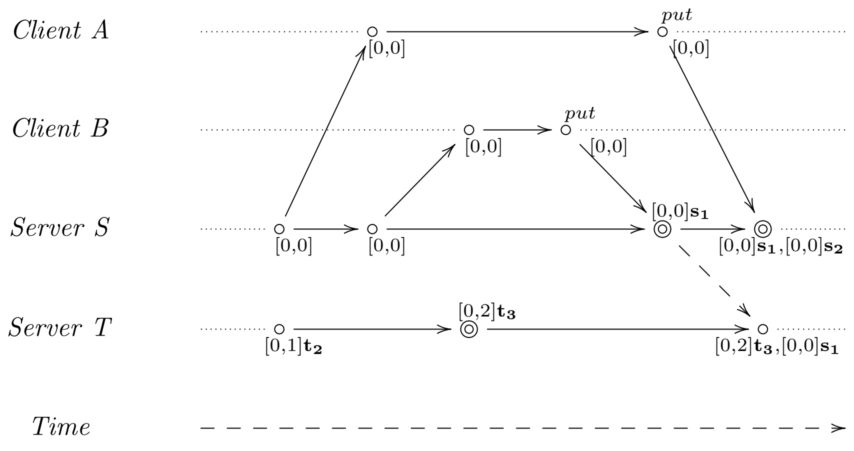


Figure 8: Run in a distributed storage system: dotted version vectors.

Figure 7 shows an example where clients A and B concurrently update server S. When client B first writes its version, a new unique name, s_1, is created (in the figure we denote this action by the symbol ◎) which is merged with the causal history read by the client, {}, leading to the causal history $\{\mathbf{s_1}\}$. When client A later writes its version, the causal history assigned to this version is the the causal history at the client, $\{\}$ merged with the new unique name, s_2, leading to $\{\mathbf{s_2}\}$. Using the normal rules for checking for concurrent updates, these two versions are concurrent. In the example, the system keeps both concurrent updates. For simplicity we omitted interactions of server T with its own clients, but we can see that before receiving data from server S it had a single version that depicted three updates managed by server T, causal history $\{t_1, t_2, \mathbf{t_3}\}$, and after that it holds two concurrent versions.

An important observation is that in each node, the union of the causal histories of all versions includes all generated unique name until the last known one: for example, in server S, after both clients send their new versions, all unique names generated in S are known. Thus, the causal past of any update can always be represented using a compact vector representation, as it is the union of all versions known at some server when the client read the object. The combination of the causal past represented as a vector and the last event, kept outside the vector, is known as a *dotted version vector* [11]. Figure 8 shows the previous example using this representation, that eventually becomes much more compact than causal histories as the system keeps running.

In the condition expressed before (clients only communicate with servers and a new update overwrites all versions previously read), which is common in key-value stores where multiple clients interact with storage nodes via a *get/put* interface, the dotted version vectors allow to track causality between the written version with vectors of the size of the number of servers.

Final remarks Tracking causality is important due to several reasons. On one hand, not respecting causality can lead to strange behaviors for users as reported by multiple authors [8, 1]. On the other hand, tracking causality is important in the design of many distributed algorithms.

The mechanisms for tracking causality and the rules used in these mechanisms are often seen as something complex [5, 13] and their presentation often lacks the necessary intuition of how they work. The most commonly used mechanisms for tracking causality, vector clocks and version vectors, are simply optimized representations of causal histories, which are easy to understand.

By building on the notion of causal histories, we believe it is simple to understand the intuition behind these mechanisms, to identify how they differ and even possible optimizations. When confronted with an unfamiliar causality tracking mechanism, or when trying to design a new system that requires it, we urge the reader to fall back to two simple questions: (a) Which are the events that need tracking? (b) How does the mechanism translate back to a simple causal history?

Without a simple mental image to guide us, errors and misconceptions become much more common. Sometimes, one only needs the right language.

References

[1] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[2] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[3] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

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

[5] Brian Fink. Why vector clocks are easy. `http://basho.com/posts/technical/why-vector-clocks-are-easy/`, January 2010.

[6] Todd Hoff. How League of Legends scaled chat to 70 million players – it takes lots of minions. `http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html`, October 2014.

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

[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] Friedemann Mattern. Virtual time and global states in distributed systems. In *Proc. Int. Workshop on Parallel and Distributed Algorithms*, pages 215–226, Gers, France, 1988. North-Holland.

[10] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, May 1983.

[11] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Brief announcement: efficient causality tracking in distributed storage systems with dotted version vectors. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 335–336. ACM, 2012.

[12] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.

[13] Justin Sheehy. Why vector clocks are hard. `http://basho.com/posts/technical/why-vector-clocks-are-hard/`, April 2010.

[14] Justin Sheehy. There is no now. *Queue*, 13(3):20:20–20:27, March 2015.

E Posters accepted for presentation

E.1 Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira. The Quest For Coordination Free Cloud Storage Systems. Accepted for presentation at SOSP'15.

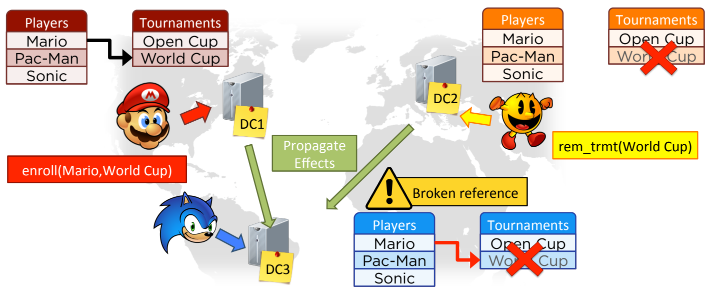
NOVA**LINCS**
LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS

THE QUEST FOR COORDINATION-FREE CLOUD STORAGE SYSTEMS

VALTER BALEGAS, SÉRGIO DUARTE, CARLA FERREIRA, RODRIGO RODRIGUES*, NUNO PREGUIÇA
NOVA LINCS, DI, FCT, UNIVERSIDADE NOVA DE LISBOA *INESC-ID / IST, UNIVERSIDADE DE LISBOA

1. Problem

- Systems need to provide low-latency and high availability for clients worldwide.
- Geo-replicated systems do not scale under Strong Consistency.
- How to ensure correctness without coordination?



2. Objective

- Eliminate global coordination completely to provide true high-availability.
- Ensure that database invariants always hold.

3. Invariant repair

- Repair invariant violations when conflicts occur.
- Classical solutions:
  - Ensure a total order of updates.
  - Requires a global vision over the database state.
  - Do not address partial replication.

4. Explicit Consistency for invariant repair

1. Specify application

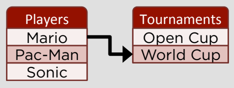
$Inv = enrolled(p,t) \Rightarrow player(p) \land tournament(t)$

enroll(p, t): {*enrolled*(p,t) := true }

removeTournament(t): { *tournament*(t) := false}

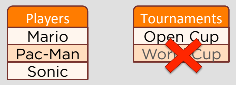
3. Choose resolution rule and transform operations

Recreate tournament



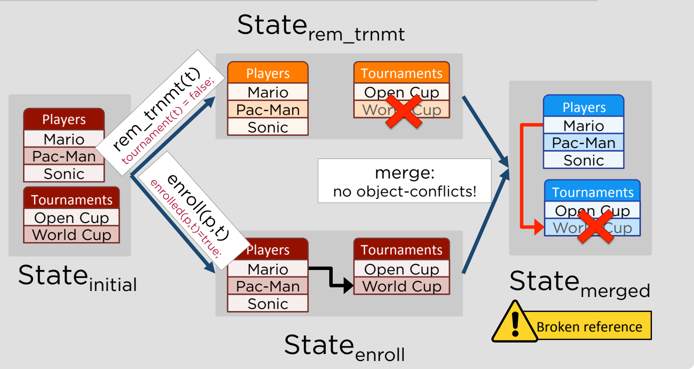
enroll(p, t): {*enrolled*(p,t) := true, tournament(t) := true}

Disenroll player

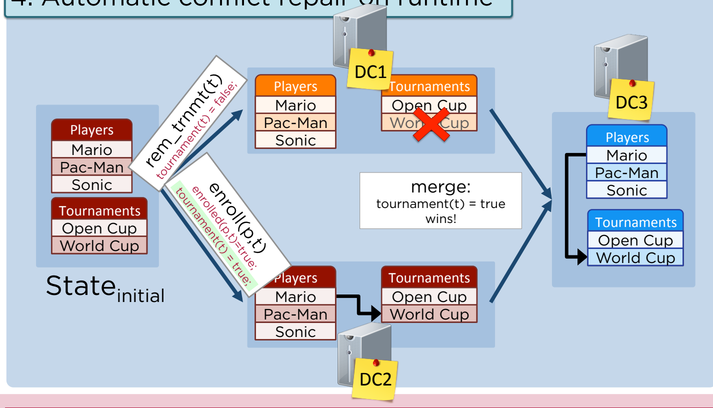


rem_trnmt (t): {*tournament*(t) := false enrolled(*,t) := false}

2. Static analysis detects conflicting operations



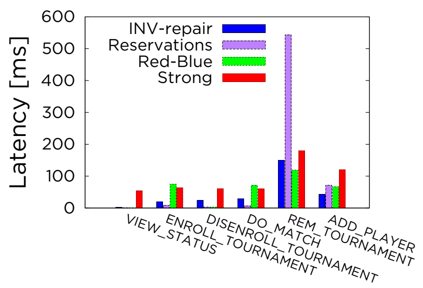
4. Automatic conflict repair on runtime



5. Preliminary results

- Comparing invariant repair to other techniques.

- 3 DCs in AWS.
- Clients submit ops to local DC.
- Compare different consistency models



6. Challenges

- How to ensure that multiple transformed operations converge and preserve invariants?
- How to address operations with dependencies?
- How to automatize operation transformation?
- Optimize transformed operations.

NOVA**LINCS** LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS
FCT FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA
FCT Fundação para a Ciência e a Tecnologia MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR
SYNC FREE

EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PTDC/ EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014.

F Papers presented without publication

F.1 Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira. Designing Concurrency-Aware Geo-replicated Storage Systems. Presented at W-PSDS'15.

Designing Concurrency-aware Geo-Replicated systems

Valter Balegas, Sérgio Duarte
Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça
NOVA LINCS/FCT/Universidade Nova de Lisboa

***Abstract*—Geo-replicated systems present a range of consistency semantics spanning from weak to strong consistency. Systems with strong consistency have prohibitive coordination requirements, therefore developers prefer using weak consistency semantics, giving up on correctness for a wide range of applications. Recent work has made possible to achieve correctness relying exclusively on weak consistency semantics, however the strategy employed requires coordination prior to operation execution to ensure safety, which might result in the system becoming unavailable when some nodes are down.**

We propose an alternative strategy that does not require safety guarantees prior to operation execution. Instead, the system allows every operation to execute locally and replicate them in a safe way, preventing application invariants from being broken when operations are delivered to remote peers. In this papers, we present our visions and discuss algorithms that can be used to provide the intended semantics.

I. Introduction

The design of geo-replicated systems requires making difficult trade-offs. In one hand we have Strong Consistency systems [9], [33] that provide correct applications but operations have high latencies and systems are less available and scale poorly. In the other hand, weakly consistent systems [10], [21], [2] are highly available, provide low-latency operations and scale better, but cannot provide the same correctness properties of the former platforms. Platforms with this level of consistency are far more popular to provide global scale systems, as liveliness and availability appear to be a factor for the success of production services [26], [1]. However, despite the performance being better, programming applications on top of these systems are an error prone task, due to the few guarantees that the model provides — data converges in the future.

To make these systems easier to program, researchers have come up with new abstractions that can improve semantics without impairing their availability and liveliness properties of these systems: replicated data types [27] provide a sensible way to handle convergence of operations without losing updates, with different semantics for different data-types; Session guarantees [30], and causality [21], [22], can strengthen the ordering of operation execution, to help preserve the intention of the users. Despite weak consistency models work well with many applications, or more precisely, only shows anomalies that are tolerable by the application, there are a wide range of applications that require Strong consistency to work properly. Systems that require Strong consistency are typically less available and operation latencies are higher, due to the fact that they require global coordination to ensure correctness.

A key insight is that not all operations in an application workload have such strong coordination requirements, therefore coordination can be used at a fine grain to ensure correctness, or otherwise, use weak consistency to improve the availability and latency of operations when possible [20], [29]. Pushing the responsibility of choosing the appropriate consistency level for the developer is cumbersome, therefore strategies to automate the choice of consistency have been developed [19], [28], [3].

Our recent work on the topic addresses the optimization of operations that require coordination. To that end, we require the identification of the fundamental invariants of the application, determining the operations that conflict with each other regarding the specified invariants and the deployment of special mechanisms capable of handling different conflicts efficiently. Our strategy combines static analysis and runtime support to extract more concurrency from operations, but still requires replicas to coordinate, out of the critical path of execution, to ensure that operations that execute concurrently do not interfere with the invariants. This coordination step makes the system less available because some operation might require coordination with a remote replica that is unavailable at the time.

In this paper, we discuss an alternative idea to ensure applications correctness without enforcing coordination. We make two important observations that drive our research: 1) Human interactions are inherently concurrent and people are aware of that. 2) Some activities need a coordinator to ensure that a procedure finishes well. When coordination is impracticable, a uncoordinated solution with confirmation step afterwards can be employed. Typically, consistency models that ensure application correctness are poorly available. We believe that to improve the availability of system, the next step is to treat conflicts as part of the application. In this paper we propose the use of compensating actions to handle conflicting executions that break application invariants. Compensating actions can be generated automatically, or can be provided by the programmer, and are applied as soon as a conflict is detected to restore the invariants of the database locally.

We assume that replicas use a concurrency control strong enough to ensure the correctness of applications, therefore invariant violations only occur when operation effects are propagated between different replicas. We handle those situations by applying a convergence strategy that repairs the invariants of the application that might have been broken by the concurrent execution, thus restoring the correctness of the application. In order to accomplish that it might be necessary to make the effects of one operation dominate the other, or apply some custom action to compensate for the occurrence. The first solution produces an outcome equivalent to a serializable execution, while the second acknowledges the mistake and applies a proper resolution, as would occur in real life. Work on compensating transactions has been studied in the past [11], [15], however existing systems do not provide programming support for that. Also, existing work does not address an important challenge of geo-replicated systems, which is to provide solutions that are coordination free to keep the systems responsive and available.

The document is organized as follows: in section two, we introduce Explicit consistency [4], a consistency model that allows us to exploit concurrency without interfering with the correctness of applications; section three presents the motivation for invariant repair; section four analyzes different repair algorithms with concrete examples; we present the work in section five and present some closing remarks in section eight.

II. Explicit Consistency

Explicit Consistency [4] is a consistency semantics where programmers identify the correctness properties that characterize an application and the systems only need to enforce those properties. Correctness properties are defined in terms of invariants — logical properties expressed over the database state. An execution trace of multiple clients across different replicas, ordered by the happens-before relation [18], is said to be I-serializable if all the possible permutations of concurrent operations, only produce database states that satisfy the invariants of the application. A system that only produces I-serializable traces satisfies Explicit Consistency. Serializability trivially satisfies Explicit Consistency. To identify I-Serializable traces one can use a static analysis tool [4], [28]. The approach taken in Indigo does so by requiring the developer to provide the specification of operations using first-order logic to express the post-conditions of the operations in the workload.

Explicit consistency is implemented in three simple steps: first, the programmer provides a specification of the application and the effects of each operation; second, a static analysis determines the pairs of operations whose concurrent execution conflict with the specified invariants; third, the application code is instrumented by the programmer, or automatically, with mechanisms to execute those operations safely and avoiding coordination.

In Indigo, the main mechanism to enforce application invariants consists in using reservations. Reservations are a family of data-types that ensures that can be used to ensure that operations only succeed locally if the correctness of the application is preserver globally. Escrow counters [24] and Multi-level locks [4] (locks with shared/exclusive access) are examples of reservations. The reservations use coordination to enforce safety prior to operation execution. This mechanism avoids coordination by amortizing the coordination costs by using a single coordination step to ensure the execution of one or multiple operations by the same replica. The downside of the approach is that it might become impossible to obtain a reservation to execute some operation of the workload if some node becomes unreachable.

In this work we explore a different approach for implementing Explicit consistency. Instead of ensuring safety prior to operation execution, we assume that local operations are always safe to execute and that conflicts only occur when the effects of an operation is delivered on a remote replica, because it is incompatible with operations executed there. When delivering the updates the system must be capable of identifying that those effects conflict with the locally executed operations and restore the application invariants.

III. Our take on the CAP theorem

Geo-replication improves the quality of service in two directions of the vector: consistency and availability. Both conditions are very desirable in the perspective of service providers, but applications are much harder to program because it is necessary to account for the execution of operations that can take an unbounded time to be delivered at all replicas. It has been widely accepted that it achieving the three goal, consistency, availability, and partition tolerance is impossible [7]. In practice, partitions are unavoidable, but not permanent, therefore the effort is to handle partitioning explicitly to achieve the best of availability and consistency.

The limitations with modern approaches to the CAP theorem is that consistency is defined in terms of ensuring an ordering of operation execution that is capable of satisfying the application invariants. Total ordering operations naturally satisfies any defined invariant, as operations effects always affect the most recent state of the database, therefore it automatically enforces application invariants. But, to enforce a total order of operations it is necessary that a quorum of replicas is always available [13] to ensure that replicas can agree on the ordering of operations. To provide better availability under partitioning, the alternative is only to enforce a partial ordering of operations, such as causality, that allows replicas to make progress without coordinating with remote peers. In this case, conflict-resolution rules are necessary to ensure data convergence [27], [29], [21], [22], [32], when updates are propagated across replicas. Conflict-resolution policies are defined on a data-types basis, which might result in application invariant violations, since those resolution do not take into account the semantics of the application. In our approach we make replication aware of the application invariants, to ensure that conflict-resolution maintains database invariants. Next we discuss the desired semantics for our conflict-resolution approach and how our approach relates to real-world scenarios.

A. *Handling the conflicts explicitly*

In the real world, companies want to maximize their revenue, they are not willing to let got some client because they cannot ensure that they can accomplish her request. Services ran by humans naturally operate in parallel: imagine two salesmen, that are both trying to sell the stock of a product, but they might be more successful then expected and outsell the product. This would be an unusual situation and they could simply contact the last buyers of the product to inform them that they ran out of stock. This might require further actions from the selling entities, but they are capable of handling this exceptional situation. In the other hand, some other sales are more sensitive, for instance, selling the item would require to make a commitment contract. In that case, the salesman would prefer to contact a manager to ask if they still have stock available, to avoid compensating for overselling the item. It is still possible that the manager becomes unavailable, for instance, because of a meeting, therefore it would be convenient that the salesmen would know how many resources they can sell without asking permission.

Humans understand that their actions affect others and we can make ad-hoc decisions to account for those situations or to prevent them. Many authors [14], [6], [12], [11] have discussed scenarios where operations may have to be retracted in order to allow better availability and parallelism, they take real world examples of applications that do the same to support their arguments. Despite the fact that we are aware that our action are not irrevocable, especially online, in practice, no modern system provides a programming support to handle them n a structured way. In the following sections we discuss concrete examples of invariant violations and discuss possible resolutions for each conflict. Later, we sketch algorithms to preserve those invariants.

IV. Tournament application

I this section we present the tournament example and a few conflicts that may arise during the concurrent execution of operations in the workload. We discuss the semantics of the execution that ensures a total and partial ordering of operations to support our convergence rules.

The tournament example is a micro-service that could be used to support most common competition online games. The operations described here are based on a previous version of the same example, first presented in [4]. We now describe the features of the application. Players participate in tournaments and compete against each other in matches. A tournament has three phases: an enrollment phase where players can enroll in the tournament, an active phase where there can be no modifications to the participants of the tournament and a finished phase, when the tournament is concluded and a winner is elected, based on the number of points achieved in each match. A tournament cannot be removed after it starts and has a minimum and maximum number of participants. A tournament has a leader that can start or remove the tournament, the leadership role can be shared with other players. A player can deposit and spend credit anytime to buy items that are used in-game to get advantage over the adversary. Items have limited availability.

A. Example 1: A matter of ordering

While a tournament does not start, players can enroll and disenroll, but the tournament can only start after a minimum number of players have enrolled in the tournament. When a partial ordering of execution is allowed, this constitutes a problem for invariant preservation: a leader of the tournament can start the tournament because he observer, in the local replica, that there is a minimum number of players enrolled, however, concurrently, at a remote replica, a player might disenroll from the tournament, dropping the number of players below minimum. Under serial execution this does not occur because one of the operations will fail, i.e., either the player cannot disenroll from the tournament, because it starts before, or the tournament cannot start because it does not have enough players. Despite the fact that serialization ensures the applications invariants, programmers need to check that the preconditions of the operations are met before modifying the state of the database.

Under partial ordering execution, the operations must also check the pre-conditions of the operations before taking any action locally, but that does not preclude a concurrent operation from interfering with this one. It might occur that a concurrent remote operation also satisfies its local dependencies but is conflicting with the current operation, and, when both operations are delivered in the same replica, an invariant violation occurs.

Different strategies to repair the invariant violation are possible: we can apply a repair function that makes none of the operations take effect; or the player is not disenrolled from the tournament and the tournament can start, or the player is disenrolled from the tournament and the tournament is canceled. The first solution does not provide a good user experience, because both users will see their actions retracted. The other two repair functions provide a semantic equivalent to the serializable execution, i.e. operations appear to have executed one after the other. However, there is an important caveat with this conflict resolution: more operations might depend on the operation being repaired, for instance, a player might have participated a match after the tournament had started and if we chose to cancel the tournament, that game should have not occurred. In this case it is easy to stop invariant violation from contaminating other operations. We can chose to remove the player from the tournament, in which case no other operation is affected by this convergence policy because no other operation in the workload depends on the player not being enrolled in the tournament to be able to execute [1].

In general, it might be necessary to analyze conflict resolution strategies in order to prevent the generation of new conflicts. We intend to study static analysis to evaluate the quality of repair strategies.

B. Example 2: When ordering is not enough

In some situations, invariant violations are not easily repaired. Consider that two players concurrently bought the last unit of an item in the application. For this conflict we cannot apply a repair function that produces a state equivalent to one operation executing after the other, because one of the requests would have different effects, i.e. the operation would fail because there are no available resources left. This situation occurs when operations are not commutative, which means that we cannot arbitrate an ordering for their execution without producing different effects. This is different from the previous example because, in the first case, despite arbitrating the execution ordering of the pair of operations, the effects of both operations are preserved.

In fact, a serial execution is what makes most sense in the real life, as it would be impossible to duplicate resources. We could think of a service that allows items to be sold in parallel and therefore overselling, but we cannot take more items then physically available.

[1]We do not require the player to be disenrolled from a tournament to allow the enroll operation because this operation is idempotent

If this invariant is important for the application, we have no option then to use a strong coordination mechanism to ensure that no user buys more resources then available. However, some invariants, or lets say, application properties, are desirable properties and not essential for correctness, in which case more solutions are possible. To not be unfair with any player, the applications could allow the item to be sold twice which is equivalent to the semantics of eventual consistency. Or, remove the item from one of the player's inventory and give back some credit. In this case, she might have used the item already and that would create more conflicts. The developer can still make this choice, as long as she is able to repair any operation that used the resource. The last alternative is to create new items to compensate for the advantage that were given to both players.

Our conclusion is that some operations naturally require a coordinated execution, but one can make an alternative version of the same algorithm that does not require serialization and apply a compensation when things go wrong. This is how online stores deal with exhausted stocks, or ATMs handle withdrawals that cannot read the actual balance of an account [6], do in practice.

V. Algorithms to repair invariant violations

In the previous section we discussed two examples of invariant violations and described possible semantics to correct them. In this section, we are going to describe different algorithms to implement those repairs.

A. *Invariant-aware convergence rules*

Consider that we repair the invariant violation of section IV-A by keeping the player in the tournament. we pursue a repair strategy that does not impair the availability of the system, therefore we avoid strategies that assume a central authority or require coordination to ensure that the invariant is repaired.

The algorithm we propose is based on the convergence rules used in CRDTs [27]. CRDTs can ensure add-/remove-wins policies when concurrent add and remove operations execute over the same data-type. This means that we can select the outcome of a concurrent add/remove operation of the same element to a set. In the example, the begin operation checks that the set of participants in the tournament has the minimum number of players and then changes the value of some flag to true, meaning that the tournament has started. The concurrent disenroll operation removes one element from the participants set, making its size smaller then the minimum and when both operations are propagated to the same replica we end in a state with a set of participants that is smaller than the required for the value of the flag being true.

The solution for this problem is quite easy. Considering that the set of participants uses a add-wins strategy for handling conflicting adds and removes. This allows to ensure that that the size of the tournament does not decrease with any concurrent remove, because we can cancel the effect of the remove with an add. In order to do that, when starting the tournament, we just add again, to the set of participants, all the players that belong to the set of player in the moment the tournament starts. This enforces that any concurrent remove will take no effect, because the merge strategy of the set preserves the concurrent adds, therefore the size of the set does not decrease. Adding all the elements to the set again can be done in an efficient way, to avoid processing overheads when the tournament is large.

The benefit of this strategy is that it does not require any additional mechanism to detect conflicts, as the execution of the operations automatically enforces the pre-conditions for the operation hold when its delivered to any replica. This strategy additionally requires identifying the pre-conditions of operations, instrument the code with the extra updates and check for compatibility between operations, i.e., that the different convergence rules are compatible with the invariants. We recognize that it might not be possible to handle all conflicts with this strategy, but it is promising.

B. *Compensating for conflicts*

In section IV-B, we describe an example where the invariant violation cannot be repaired, thus the system must handle it as part of an exception of the workload. To handle those situations we want to apply some action that compensates for the occurrence. In the previous example, the solutions consist in doing nothing, remove the item from one player's item list, or create new resources. However, two things have to be taken into consideration when writing compensations: does the compensation conflict with any other invariant? what happens if two different replicas compensate the same action? To answer the first question we can consider a compensating action as part of the workload and use the same tools to detect the conflicts of the application. Applying compensating actions is trivial when the operations are idempotent, because they can execute at multiple machines without producing further outputs. The problem with applying the compensating actions is that if when the operation is non-idempotent, in which case, multiple executions of the same operation produce cumulative effects. For instance, if the compensation was to create new resources, it could create more resources than desired.

The simplest way to implement the compensation mechanism is to use a central authority that would guarantee that the compensating action only occurred once, or use a consensus algorithm. However, this requires coordination which is what we are trying to avoid at all costs. The alternatively is to make compensations idempotent. To enable that, replicas need to maintain the log of the operation they applied, the information of what compensations they applied to solve each conflict and the compensating operations must be deterministic and independent of the current state of the database. If every replica keeps this information, they can independently identify what remote replicas have applied the compensation and cancel the effect of multiple repairs. The downside of the approach is that replicas cannot compress the log until all replicas have received the conflicting updates, but we can assume that partitions, when they occur, do not last forever. The damage of compressing the log before a replica acknowledges a conflict is measurable and the developer can decide to move forward after some time, to avoid the log size to increase. The result would be that the unreachable nodes could have compensated for the same conflict and the effects will accumulate. Another property of compensating transactions is that they may not have to be executed immediately, i.e., the system can delegate applying the fix to the future, which can be convenient in some cases.

VI. Related work

Geo-replicated systems are at the core of Cloud infrastructures. Existing system provide a wide spectrum of consistency levels, from systems that provide Weak Consistency [10], [8], [21], [22] to Strong Consistency [9], [23], [16] or a combination of both [20], [29]. Eventual Consistency is a very popular consistency model in production environments. The wide range of industry platforms that implement it is a proof of that [5], [17]. Not all application work well under Eventual Consistency. Researchers have tried to reach the boundaries of Eventual consistency: CRDTs [27] explore commutativity for enabling automatic merge of concurrent operations; Causal Consistency [21], [22] ensures that operations ordering preserve the intention of the users; Bailis et al. have studied what invariants can be provided with high availability and when coordination is required to ensure correctness [3]. Quelea [28] is a tool that capable of generating a protocol that enforces the minimum consistency semantics that is necessary to maintain the correctness of an application.

While the limits of Weak consistency and Strong consistency are well understood, other work tries to explore the grey area between the two semantics. Indigo [4] is capable of providing Strong consistency semantics relying exclusively on weak consistency protocols. To that end, the system combines analysis techniques and runtime support to ensure what operations are safe to execute without coordination. The homeostasis protocol [25] is also capable of ensuring safety with operations executing locally, but requires two-phase commit to communicate between replicas.

Many authors share the vision that better availability properties are only possible with weaker transactions/consistency semantics [11], [6], [14]. When correctness properties ave violated, a compensation action can restore the invariants of the database [11], [15]. Bayou [31] puts these ideas into practice. In Bayou, replicas can execute operations without coordination and conflicts are merged after being detected with custom merge operations provided by the programmer. In Bayou conflict resolution can only be applied at object level, similarly to CRDTs. In our work, we revise this approach in the context of geo-replicated system and generalize the idea to operations that touch multiple objects. We propose fixing conflicts without using any conflict detection mechanism, to preserve the availability of the system. We envision that repair functions can be generated automatically from the application specification. When a custom action is preferable, we check that this action does not create any new invariant violations.

VII. Conclusion

In this paper, we proposed new ideas to improve the availability of systems working on top of geo-replicated Cloud storage. Our insight is that applications need to deal with conflicts explicitly to improve correctness without loosing availability. The solutions we proposed allow the system to remain correct as long as operations exhibit some desirable properties that allow conflicts to be repaired. When conflict are not repairable, the solution is to fall back to coordinated execution, implement the operations with weaker semantics, or acknowledge the conflict and provide an operation to compensate the occurrence. We briefly discussed algorithms to repair invariant violations and apply compensating transactions that have good availability properties.

References

[1] Abadi, D. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer 45*, 2 (Feb. 2012), 37–42.

[2] Almeida, S., Leitão, J. a., and Rodrigues, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 85–98.

[3] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination-avoiding database systems. *CoRR abs/1402.2237* (2014).

[4] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N. M., Najafzadeh, M., and Shapiro, M. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015* (2015), p. 6.

[5] Basho. Riak. *http://basho.com/riak/*, 2014. Accessed Jan/2014.

[6] Brewer, E. Cap twelve years later: How the "rules" have changed. *Computer 45*, 2 (2012), 23–29.

[7] Brewer, E. A. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2000), PODC '00, ACM, pp. 7–.

[8] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1277–1288.

[9] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[10] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[11] Garcia-Molina, H., and Salem, K. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1987), SIGMOD '87, ACM, pp. 249–259.

[12] Gray, J. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (1981), VLDB '81, VLDB Endowment, pp. 144–154.

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

[14] Helland, P., and Campbell, D. Building on quicksand. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings* (2009).

[15] Korth, H. F., Levy, E., and Silberschatz, A. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1990), VLDB '90, Morgan Kaufmann Publishers Inc., pp. 95–106.

[16] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. Mdcc: Multi-data center consistency. In *Proceedings of*

the 8th ACM European Conference on Computer Systems (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 113–126.

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

[18] LAMPORT, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.

[19] LI, C., LEITÃO, J. A., CLEMENT, A., PREGUIÇA, N., RODRIGUES, R., AND VAFEIADIS, V. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 281–292.

[20] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 265–278.

[21] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. 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* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[22] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 313–328.

[23] MAHMOUD, H., NAWAB, F., PUCHER, A., AGRAWAL, D., AND EL ABBADI, A. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow. 6*, 9 (July 2013), 661–672.

[24] O'NEIL, P. E. The escrow transactional method. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986), 405–430.

[25] ROY, S., KOT, L., BENDER, G., DING, B., HOJJAT, H., KOCH, C., FOSTER, N., AND GEHRKE, J. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), pp. 1311–1326.

[26] SCHURMAN, E., AND BRUTLAG, J. Performance relatedchanges and their user impact. Presented at velocity web performance and operations conference, 2009.

[27] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.

[28] SIVARAMAKRISHNAN, K., KAKI, G., AND JAGANNATHAN, S. Declarative Programming Over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN conference on Programming language design and implementation* (2015), PLDI '15.

[29] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 385–400.

[30] TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems* (Los Alamitos, CA, USA, 1994), PDIS '94, IEEE Computer Society Press, pp. 140–150.

[31] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 172–182.

[32] ZAWIRSKI, M., BIENIUSA, A., BALEGAS, V., DUARTE, S., BAQUERO, C., SHAPIRO, M., AND PREGUIÇA, N. M. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *CoRR abs/1310.3107* (2013).

[33] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 276–291.