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.2.2.1<br>CRDTs and CRDT composition<br>in partial-replication setting |
| Due date of deliverable: | April 1, 2015 |
| Actual submission date: | March, 2015 |
| Start date of project: | October 1, 2013 |
| Duration: | 36 months |
| Name and organisation of lead editor for this deliverable: | Technische Universität Kaiserslautern |
| Revision: | 0.1 |
| Dissemination level: | PU |

# Contents

# 1 Executive summary

The SyncFree project aims to enable large-scale distributed applications in geo-replicated settings. To this end, we rely on replicated yet consistent data types (CRDTs), which allow information dissemination and sharing without the need for global synchronization.

Within the project, Work Package 2 (WP2) develops the core protocols and algorithms for CRDT data stores in different topologies. The focus of the first year of the project has been on architectures with full replication at the server side. This scenario assumes a small number of replicas, a mostly-static topology, rare failures, and powerful servers.

Moving towards highly-scalable distributed systems, the second deliverable for WP2 puts a focus on mechanisms for scaling distributed data stores to a large number of nodes. *Partial replication* plays a central role hereby: With the increasing growth of data that is accumulated and transferred between nodes for high accessibility and scalability, full replication for all data items becomes unfeasible eventually – in particular, if data is not only replicated at DCs, but also at clients.

Hence, replicating only a subset of data items at each node results in less requirements regarding local space and computational power. However, partial replication raises a number of questions which we address with our work. We give here a brief overview of the results achieved during the reporting period.

**Causal consistency under partial replication** Causal consistency provides a weak, but natural and expressive notion of consistency in replicated data stores. Under causal consistency, updates can occur concurrently at different nodes and are then propagated asynchronously to the other nodes in the system. An update becomes observable at a node once the causal history has been delivered at this node. Due to the transitivity of the causality relation between updates, this locally applied criterion might involve data items that are not replicated locally. We have developed and implemented a protocol for causal consistency under partial replication which allows for genuine partial replication. The key idea is to perform dependency calculations at an origin DC based on the knowledge about the receiver DC. Only when the origin DC has confirmation that the receiver DC has obtained the causal dependencies for an update, this update is propagated. This scheme reduces the amount of dependency checking, while slightly increasing the visibility latency of updates at other nodes.

**Conflict-free Partially Replicated Data Structures** Replicating objects not only on the server side, but also at clients asks for further refinements of partitioning schemes. Instead of (vertically) dividing a data store into smaller groups, with possibly overlapping subsets of data items, the data items themselves can be subject to partitioning. Some of the replicated data items in our context, i.e. CRDTs, can grow quickly in size. For example, a Set CRDT grows (at least) linearly in the number of its elements. However, in many practical situations, users will be only interested in few entries from a set; if the Set CRDT represents the posts from a user in a social network, only the latest entries should be replicated to the client, the other entries only on demand. In our work on Conflict-free Partially Replicated

Data Structures (CPRDTs), we examined the potential of CRDTs for partitioning of the payload and introduced a specification format for CPRDTs based on particles, that constitute the basic building blocks for CPRDTs. We specified a few CPRDTs and successfully evaluated a prototype implementation in a vote-based content-sharing application.

**Adaptive replication** Under partial replication, data items are only replicated at a selection of the nodes in the system. Choosing the nodes where a specific data item is replicated has a huge influence on the access latencies and overall performance of the data store. In a complimentary line of work, we investigated the replication distribution such that replicas reside in DCs close to users while reducing the network traffic and space requirements for keeping replicas up-to-date and highly accessible. We have developed and implemented a prototypical component for adaptive replication. The basic idea behind our approach is that a high (relative) number of reads and write issued through a DC indicates a suitable place for a replica (assuming that clients connect to the DC closest to them in order to reduce latency). Each replica is associated with a strength factor that is influenced by a number of factors (local read and writes, decay factor, writes to other DCs). If the strength factor reaches an installation threshold, a replica is locally installed; if it falls below the replication threshold, the local replica is removed, subject to additional conditions such as guaranteeing availability of a minimal number of replicas.

# 2 Milestones in the Deliverable

WP2 has reached the following milestones:

| Mil. no | Milestone name | WP | Date due | Actual date |
|---|---|---|---|---|
| S1 | CRDT consolidation in a static environment | WP2 | M12 | M12 |
| S2 | Extended guarantees and composition in a dynamic environment | WP2 | M24 | M24 |

The corresponding tasks are:

| Task no | Task name | Date due | Actual date | Leader |
|---|---|---|---|---|
| D.2.1.1 | Protocols for CRDTs in small-scale full replication | M6 | M12 | KL |
| D.2.1.2 | Platform for CRDTs in small-scale full replication | M6 | M12 | KL |
| D.2.2.1 | Protocols for CRDTs and CRDT composition in partial-replication setting | M12 | M18 | KL |
| D.2.2.2 | Platform for CRDTs and CRDT composition in partial-replication setting | M12 | M18 | KL |

**Shifting of milestones** Several of the main developers on WP2 could only be recruited and employed in February 2014, accounting to the delay of several months. To allow for integration of tools and libraries provided by the industry partners (e.g. riak_core, riak_bench, Quickcheck), we chose Erlang with its Open Telecom Platform (OTP) as programming language and development platform. This led to another delay of some weeks as the developers were not familiar with Erlang initially. Thus, the design and development of Antidote has started effectively in March/April 2014. The M6 and M12 deliverables for WP2 have therefore been moved by 6 months. The executive board approved of this adjustment of the deliverable dates.

# 3 Contractors contributing to the Deliverable

The following contractors contributed to the deliverables

## 3.1 KL

Annette Bieniusa, Deepthi Akkoorath.

## 3.2 INRIA

Alejandro Tomsic, Tyler Crain, Marc Shapiro.

## 3.3 Louvain

Manuel Bravo, Zhongmiao Li.

## 3.4 Nova

Diogo Serra, Nuno Preguica, Valter Balegas.

## 3.5 Trifork

Amadeo Ascó.

# 4 Results

In this section, we present the results obtained for Task 2.2 during month 12 – 18. Our research concentrated on two topics: causal consistency under partial replication and adaptive replication strategies. We first give an overview on challenges and issues that arise under partial replication and elaborated our results in subsequent sections.

## 4.1 Partial replication

The amount of data being processed in DCs keeps growing at enormous rate. At the same time, DCs and data stores are moving closer towards the client in order to reduce latency and increase accessibility. The resulting increase in the number of replicas incurs additional cost in terms of coordination and memory usage. Replicating all data items in all distributed instances of a data store thus will become unfeasible.

*Partial replication*, i.e. replicating only subsets of the data items at a DC, is a possible solution to approach this problem. Implementing such a solution is particularly difficult when the system makes the placement of replicas transparent and hides this complexity from the user.

Partial replication raises a number of challenges:

- *Replication strategy*: Where should replicas of some data item be placed? How many replicas should exist in the system? How can replicas be located and accessed?

- *Update dependency tracking*: How can the system prevent users observing inconsistencies in the data? How can atomic updates across multiple keys be performed when not all keys are present in a node? What meta-data can encode the dependency of updates in an efficient and scalable manner? How must updates be structured such that only the parts that are relevant for a replica are forwarded and applied to this replica?

In our work, we addressed both challenges on different levels. In the following sections, we discuss the aspects mentioned before and how we addressed them with in our work.

## 4.2 Causal consistency

Causal consistency has proven to be the strongest consistency model under which low-latency and high-availability can be achieved [28]. In addition, this model is easier to reason about for programmers than eventual consistency, its previously widely-adopted weaker counterpart.

In our work, we focus therefore on *causal consistency* as the preferred consistency notion. Under causal consistency, updates can occur concurrently at different nodes and are then propagated asynchronously to the other nodes. To reflect the (potential) causality of updates, the reads-from order and session order between reads and writes is tracked. An update is only observable at a node once the causal history of the update has been delivered at this node.

In the past few years, many causally consistent systems have been developed [24, 22, 18, 5, 14]. These systems differ in their implementation due to the assumptions and compromises they make. For instance, there are protocols that track potential dependencies [24, 22, 18, 14]; defined by the happens-before relation [20] between events, while others just track explicit dependencies [5]. Another important trade-off is visibility latency vs. throughput [18]. In this line, most protocols use explicit dependency check messages; which improves visibility [24, 22, 18, 5, 1, 14] while others improve throughput by utilising a stabilisation mechanism [18] that slightly penalises it.

Choosing among a large number of systems that provide causal consistency can be hard. Even when protocols are well documented, the used vocabulary, naming conventions and perspectives vary. Moreover, design considerations, topology assumptions and implementation differences further constrain the possibility of a fair comparison. Finally, most protocols only compare to a few alternatives and/or to an eventually- or strongly-consistent baseline. It thus remains complicated to understand the important differences among causally-consistent protocols and to make an objective, scientific comparison of their behaviour.

To gain a better understanding of the characteristics for the protocol, we have started with a comparative study of the different implementations of causal consistency. In recent work, Saeida Ardekani et al. [31] identified that there is a family of strongly-consistent protocols that share a generic algorithmic structure, called DUR (deferred update replication). Briefly, DUR protocols execute transactions in two phases: an execution phase, were values are read and updates are buffered; and a termination phase, where an atomic commit protocol decides on the outcome of a transaction, and its effects are propagated across the system. Their work introduced the G-DUR framework, a tool for implementing DUR protocols, and an empirical comparison of well-known strongly consistent systems. Interestingly, causally-consistent protocols also present a DUR structure. We also observed that most implementations of causal consistency fall into a sub-category of DUR, Asynchronous-DUR (ADUR).

A-DUR protocols present particular properties, namely: (*i*)they are topology (data center) aware; (*ii*)transactions execute and commit locally: communication only involves local replica(s) of each updated object (normally, the one(s) located at the DC where the transaction is started); (*iii*)they only incur a termination phase in the case of atomic writes, which never aborts a transaction, and, most distinctively; (*iv*)they perform *background asynchronous processing*, which handles tasks like propagating committed updates to remote DCs; checking dependencies, resolving conflicts (i.e., causal+ convergence) and applying updates at remote DCs; and/or making updates visible.

We have analysed two well-known causally-consistent protocols, Eiger [25] and GentleRain [18], and plan to extend the analysis to other protocols [35]. To simplify and stream-line comparison of these protocols, we will extend together with WP5 the G-DUR framework [3] for causally-consistent protocols.

## 4.3 Causal consistency under partial replication

Ensuring (causal) consistency under partial replication usually requires additional communication between nodes which do not host a replica of some data item, thus impeding scalability [31].

While protocols ensuring causal consistency are generally efficient when compared to strongly consistent ones, they usually do not support partial replication in a genuine way. Given the asynchrony of the system, updates might arrive in different order at some replicas. Therefore, a dependency check must be performed before they are applied to ensure the consistency of a client's view. This check is based on order-inducing metadata that is propagated along with the updates or through separate messages [7].

A classical approach to causality checking is based on vector clocks [20] where each participating node in the system is given a vector entry, and each of the updates initiated through this node is assigned a unique increasing scalar value.

Since geo-replicated systems often involve a large number of nodes, these systems build different, optimized representations of such vector clocks. For example, certain protocols use vectors with one entry per DC [39], or one entry per partition of keys [15], or vectors that can be trimmed based on update stability or the organisation of the partitions of keys across DCs [15]. Different from vector clocks, there exist other approaches including real time clocks [17], or tracking reads of memory locations [23] or operations [26] up to the the last update performed by this client.

For correctness, all of these mechanisms create (over-)approximations of the dependencies of each operation, i.e. from this metadata it is not possible to deduce exactly what the causal dependencies for an operation are. For example, when using a single vector entry per server, all operations from separate clients connected to this server will be totally ordered even if they access disjoint sets of data.

The causal relation of operation is approximated primarily because precise tracking of dependencies would not scale as the size of the metadata would grow up to the order of the number of objects or clients (depending on how dependencies are tracked). However, the issue with over-approximating dependencies is that the dependency check might lead to increased read latency. False dependencies can introduce waiting on updates to be delivered than would be necessary for correctness. Clients might therefore either be blocked or provided with stale data.

**Approximating dependencies** This approximate tracking of dependencies also creates an unintended effect of making partial replication more costly.

To see why this happens, consider the example shown in figure 1 where a protocol is used that tracks dependencies using a version vector with an entry per server. Assume this protocol supports partial replication by only sending updates and metadata to the servers that replicate the concerned object. In this example there are two DCs: $DC_A$ and $DC_B$ each with 2 servers: $A_1$ and $A_2$ at $DC_A$ and $B_1$ and $B_2$ at $DC_B$. Server $A_1$ replicates objects $x_1$ and $x_2$, server $B_1$ replicates objects $x_2$ and $x_3$ while server $A_2$ replicates objects $y_1$ and $y_2$ and server $B_2$ replicates objects $y_2$ and $y_3$. The system starts in an initial state where no updates have been performed. Consider then that a client which performs an update $u1$ on object $x_1$ at server $A_1$, resulting in the client having a dependency vector of $[1, 0, 0, 0]$,

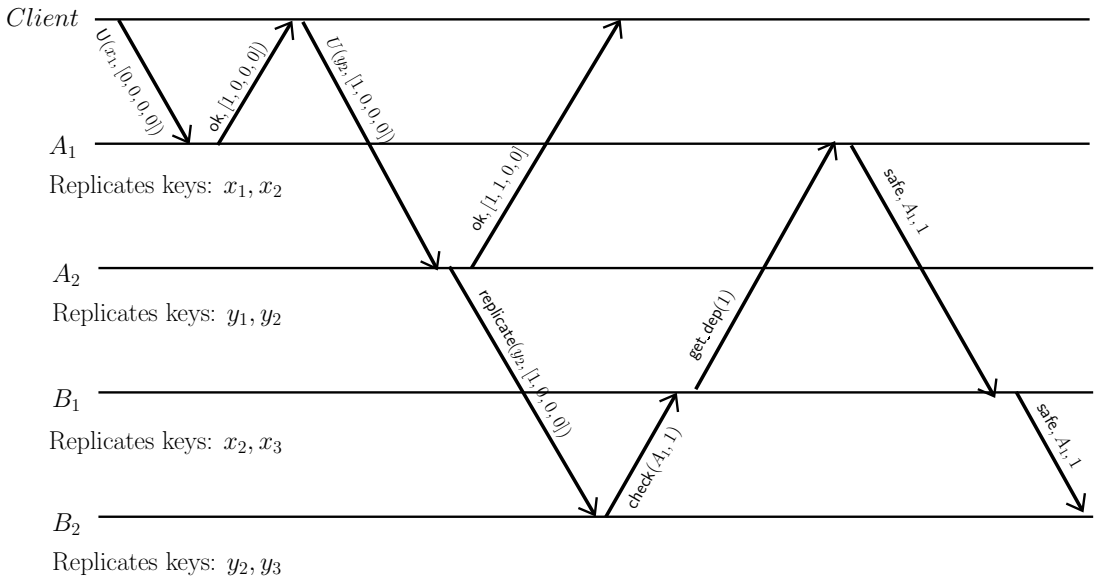



Figure 1: An example showing the dependency checks needed in a system with partial replication using version vectors with one entry per server for tracking dependencies.

with the 1 in the first entry of the vector representing the update $u1$ at $A_1$. Since $x_1$ is not replicated elsewhere, the update stays locally at $A_1$. Next, the client performs an update $u2$ on object $y_2$ at server $A_2$, returning a dependency vector of $[1, 1, 0, 0]$. The update $u2$ is then propagated asynchronously to $B_2$, where upon arrival a dependency check is performed. Since the dependency vector includes a dependency from $A_1$, before applying the update, $B_2$ must check with $B_1$ that it has received any updates covered by this dependency in case they were on a key replicated by $B_1$. This situation could occur when $B_1$ has received an update, but has not propagated the new dependency vector to $B_2$, yet.

But since $B_1$ has not heard from $A_1$, it does not know if the update was delayed in the network, or if the update involved an object it does not replicate. Thus $B_1$ must send a request to $A_1$ checking that is has received the necessary update. $A_1$ will then reply that it is safe because $u1$ did not modify an object replicated by $B_1$, which will then be forwarded to $B_2$ at which time $u2$ can be safely applied. Notice that if dependencies were tracked precisely, this additional round of dependency checks would not be necessary as the dependency included with $u2$ would let the server know that it only depended on keys not replicated at $DC_B$.

While this is a simple example that one could imagine easily fixing, different workloads and topologies can create complex graphs of dependencies that are not so easily avoided. Furthermore, current protocols designed for full replication do not take any additional measures specifically to minimize this cost. Instead they suggest to send the meta-data to every DC as if it was fully replicated, either in a separate channel [7] or simply without the update payload. In effect, those protocols do not use any specific design patterns to take advantage of partial replication.

We developed a protocol, Charcoal, to support causal consistency under partial replication in a scalable and efficient way by minimising dependency metadata and

checks. The Charcoal protocol builds on a number of mechanisms already found in many causally consistent protocols including the base version of Antidote, except here they are extended to efficiently support partial replication.

- **Update identification** Vector clocks are the most common way to support causality. To avoid linear growth of vector clocks in the number of (client) replicas, we apply a similar technique as in the work of Zawirski et al.[39] which is also used in the full replication version of Antidote. Each entry in a vector represents a DC, or more precisely a cluster within a DC. Modified versions of protocols such as ClockSI [16] or a DC- local service handing out logical timestamps, such as a version counter, can be used to induce a total order to the updates issued at this DC, which can then be represented in the DC's vector entry. Since the total ordering of updates is already provided by the full replication version of Antidote, Charcoal reuses this part of the protocol with small modifications to transparently support reads and writes to non-replicated keys (described in the following paragraphs).

  For causality tracking, each update is associated with its unique timestamp given by its home DC, plus a vector clock describing its dependencies. To provide session guarantees such as causally consistent reads when interacting with clients, the client keeps a vector reflecting its previously observed values and writes. The system then ensures that clients may only read values containing all dependencies given this vector. In our work on Swiftcloud [39], we provide further details on extending session guarantees to the edge of the system, thus ensuring consistency even when clients fail over to different DCs.

  Given that in Charcoal a DC might not replicate all objects, certain reads will have to be forwarded to other DCs where the object being read is replicated. The receiving DC then uses the client's dependency vector to generate a consistent version of the object that is then forwarded to be cached at the DC the client is connected to.

- **Disjoint safe-time metadata** In general, most protocols ensure causal consistency by not making updates from external DCs visible locally to clients until all updates causally preceding it have been received. In the fully replicated version of Antidote given that objects are organized in fixed partitions across DCs, an external update is visible when each local server has received the updates from its corresponding server at the external DC up to the dependent time. Unfortunately this cannot be done in the same way in Charcoal, as in order to support partial replication each server and each cluster or DC can replicate or not any partition of objects, thus requiring a different method for making updates visible. A simple modification to the base Antidote protocol to support partial replication would be to have each server to additionally send dependency meta-data to every server whose set of replicated objects intersects with its own set (see Figure 1 for an example of why these additional checks would be needed), but this would result in an increased cost of running the protocol compared to full replication, as locating an object requires additional communication steps.

To avoid these additional dependency checks and meta-data, the key insight in Charcoal is to perform the dependency calculation at the origin DC and not the receiving DCs. Updates are still sent directly to the other sibling replicas at other DCs, but they are not made visible to readers at the receiving DC until the origin DC confirms that its dependencies have been received. At the origin DC, updates issued up to a time $t$ are considered safe to apply for a receiving DC when all the local servers have sent all their updates on replicated data items up to time $t$. To keep track of this, a server at the origin DC communicates with each local server, keeping track of the time of the latest updates sent to external DCs. Once it has heard from each local server that time $t$ is safe, this information is propagated to the external DCs as a single message, thus avoiding unnecessary cross-DC dependency checks and meta-data propagation, saving computation and network bandwidth. Further, transaction are never forced to block and wait when reading as data only becomes visible after all causal dependencies have been satisfied.

The negative consequence of this is that the observable data at the receiving DC might be slightly more stale than in the full replication case because the receiving DC has to wait until the sending DC has let it know that this data is safe. Such a delay can be seen as a consequence of tracking dependencies approximately as seen in the example in Figure 1

- **Local writes to non-replicated keys** Given that causal consistency allows for concurrent writes, in order to ensue low latency and high availability a DC will accept writes for all objects, including those that it does not replicate. Using the vector clocks and metadata as described above this can be done without any additional synchronisation by just assigning unique timestamps to these updates that are reflected in the vector of the local DC. These updates can then be safely logged and made durable even in the case of network partition.

- **Atomic writes and snapshot reads** Beyond simple key-value operations, we provide a weak form of transactions which allows to group reads and updates together and supporting CRDT objects. Just like in the full replication version of Antidote, atomic writes can be performed at the local DC using a 2-phase commit mechanism without contacting the remote replicas in order to allow for low latency and high availability. Following this approach, the updates are then propagated to the other DCs using the totally ordered dependency metadata described previously ensuring their atomicity. (Note that atomic updates can include keys not replicated at the origin DC.) Causally consistent snapshot reads can be performed at a local DC by reading values according to a consistent vector clock, while read requests for data items not replicated at the local DC are forwarded to another DC using the same vector clock.

Using these mechanisms allow partial or full replication with causal consistency while limiting the amount of redundant inter-DC metadata traffic. All DCs are able to accept updates to any key, and causally consistent values can be read as long as one replica is available and reachable. Additionally, the way the keys are

partitioned within a DC is transparent to other DCs, allowing this information to be maintained locally.

Finally, it is important to note that while the Charcoal protocol helps mitigate some of the costs of implementing partial replication in previous protocols, it does not provide an optimal solution to the problem. Charcoal still employs imprecise representation of dependencies which can lead to false dependencies (that are checked locally within the sending DC) and can result in reading stale values not including updates that might actually be safe to read. Further research needs to be conducted in order to see whether these costs can be avoided entirely.

### 4.3.1 Implementing partial replication in Antidote

Charcoal has been designed to exist within the same framework as the full replication protocol that had already existed in the Antidote platform. While certain components needed to be reimplemented, much of the core infrastructure is reused, thus allowing the development of the platforms to not entirely diverge.

The main elements that had to be reimplemented were those that dealt with dependency generation and validation and inter-DC replication. Additionally several new components had to be developed including those to check where objects are replicated and those to collect the meta-data at the local DC concerning the safety of the update. Within a data center the components for executing transactions and materializing objects were reused with small modifications to support reading and writing to objects not replicated in the DC. Details regarding the platform and the implementation can be found in the report accompanying the deliverable D2.2.2.

## 4.4 Adaptive replication

Most works in partial replication focus on the protocols and DC interaction, i.e. on keeping the meta data and communication overhead small. The placement of replicas is often assumed to be fixed a priori and to not change over time. State-of-the-art systems [19] typically use consistent hashing to infer a random placement strategy. This limitation renders the direct applicability of these protocols in practical real-world settings difficult.

Complementary to our work on causality tracking under partial replication, we are therefore investigating adaptive replication strategies. Adaptive replication deals with dynamic replication strategies where the creation and deletion of replicas is directed by the users' access pattern to the data [12]. An optimal replication strategy minimizes the cost associated with network traffic and additional storage while keeping the access latency for users low. Most proposals for replication strategies focus on performance aspects, such as improving load balancing or reducing network congestion [30], not taking non-functional aspects into account.

In practice, the replication scheme often is subject to additional constraints. For example, the system should always provide a minimal number of replicas to improve accessibility when the system is partitioned. Another practical constraint might be that data items must not be replicated on certain nodes placed in other countries due to legal restrictions.

The problem of finding an optimal replication schema in an arbitrary network

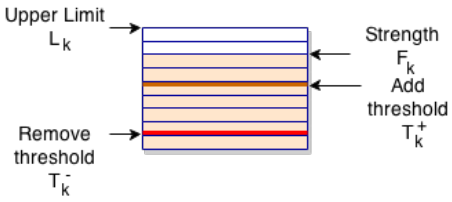



Figure 2: Replication strength and thresholds.

has been shown to be NP-complete for the static case [2, 37, 36]. Hence, there is not known an efficient algorithm for locating a convergent optimal solution for georeplicated systems. Given this theoretical limitation, we concentrate on a practically oriented solution which requires little computational overhead. We propose a protocol based on Wolfson et al.'s adaptive algorithm [36] for replicating data in a distributed system. The algorithm takes into account the changes in the read-write pattern of the users in the network. It is based on the principles of Ant Colony Optimisation algorithms, which are inspired by the behaviour of ant colonies when deciding which path to follow when foraging [13].

**Model** We developed a formal system describing the algorithm in a precise way. Each data item $k$ in a data center $d$ is associated with some strength factor $F_{kd}$. A data item is replicated in a data center when its strength exceeds a replication threshold $T_k^+$, and is subject to removal when the strength drops below a deletion threshold $T_k^-$. In the latter case, additional constraints are taken into account, e.g. a minimum number of replicas should always be retained to prevent data loss. The strength function is given by the following equation:

$$F_{kd} = \max\left(0,\ \min\left(\underbrace{L_k}_{\text{max. strength}}, \underbrace{r_{kd} * \Delta r_k}_{\text{own reads}} + \underbrace{w_{kd} * \Delta w_k}_{\text{own writes}} - \underbrace{\sum_{i=1, i\neq d}^{|DC|} w_{ki} * \Delta w_{kdi}}_{\text{other writes}} - \underbrace{X_{kd} * \Gamma_k}_{\text{time decay}}\right)\right)$$

Here, $R_k$ denotes the total number of replicas of data $k$ which must be at least $N_k$ ( minimum number of replicas). $X_{kd} = 1$ represent the existence of a replica of some data item $k$ in DC $d$, or its absence when $X_{kd} = 0$.

Interpreting this equation, the replication strength for the data $k$ in DC $d$ is increased by the number of reads ($r_{kd}$) and writes ($w_{kd}$) issued at DC $d$, with intensities $\Delta r_k$ and $\Delta w_k$, respectively. It is weakened by the writes requested through other DCs ($w_{ki}$) with intensity $\Delta w_{kdi}$. The strength is furthermore weakened by a temporal decay factor $\Gamma_k$.

The decay factor $\Gamma_k$ ensures that in absence of writes, if the reads are concentrated in a few DCs, then the replicas in other DCs will eventually be removed. However, this removal must comply with the minimum number of replicas. Hence, the decision about removing a replica once its strength goes below the replication threshold depends not only on the strength, but on a number of additional factors. Each DC must know about other DCs hosting replicas of the same data item in order to forward updates or read requests or remove local replicas.

**Protocol** In this model, only DCs hosting a replica of the data will be penalised by writes to other DCs and the pass of time. This reflects the cost of sending updates to the DCs hosting a replica which should be minimised by the strategy. Reads may increase the number of replicas while writes may strengthen the replication in a DC, but may also potentially remove a replica from one of the other DCs, an effect that is amplified by the temporal decay of the replication strength.

A read request to a DC, which does not have a replica of the data, will be forwarded to the closest DC with a replica. The DC with a replica will not gain strength from this read operation as the read was not initiated at this DC. The DC where the read request originated from will then gain some knowledge about the data access pattern even if it does not have a replica. This knowledge will be used in subsequent reads/writes to the DC and eventually may lead to the placement of a replica. Otherwise, the record will be discarded. Once the replication strength is higher than the replication threshold ($T_k^+$), this DC will notify all the DCs about the existence of the new replica on this DC.

Information about the writes at other DCs is communicated between all DCs at regular intervals, and can be even piggy-backed with update messages or forwarded read requests.

The placement information is handled by an additional component which must provide stronger consistency guarantees than the data store itself. When installing a new replica, other DCs must be notified about the new replica such that updates are forwarded also to this new replica. The information about new replica installments can be maintained with weak consistency guarantees, such as eventual consistency, as missing updates just increase staleness of data. On the other hand, removing replicas requires strong consistency if a minimal number of replicas must be guaranteed. However, this operation is not on a critical path and does not block the data store.

**Cost estimation** For our model, we also derived a cost estimation for a time interval $(t_i, t_{i+1})$.

$$
\begin{aligned}
&Cost_k(t_i, t_{i+1}) = \\
&\sum_{d=1}^{|DC|} \Bigg( \underbrace{r_{kd}(t_i, t_{i+1}) * c^r_{kd}(t_i) + w_{kd}(t_i, t_{i+1}) * c^w_{kd}(t_i)}_{\text{direct cost}} + \underbrace{\sum_{j=1}^{|DC|} w_{kd}(t_i, t_{i+1}) * c^w_{kdj}(t_i) * X_{kj}(t_i)}_{\text{update propagation cost}} \\
&+ \underbrace{(1 - X_{kd}(t_i)) * r_{kd}(t_i, t_{i+1}) * \min_{j \in R_k}(c^r_{kdj}(t_i))}_{\text{redirect reads cost}} + \underbrace{(1 - X_{kd}(t_i)) * w_{kd}(t_i, t_{i+1}) * \min_{j \in R_k}(c^w_{kdj}(t_i))}_{\text{redirect writes cost}} \\
&+ \underbrace{\sum_{j=1, j \neq d}^{|DC|} (1 - X_{kd}(t_i)) * X_{kj}(t_i) * \rho_{kd}(t_i, t_{i+1}) * c^r_{*dj}(t_i)}_{\text{search cost}} \\
&+ \underbrace{\sum_{\substack{j=1, \\ X_{kj}(t_i)=1}}^{|DC|} X'_{kd}(t_i) * (1 - X_{kd}(t_i)) * c^r_{*kdj}(t_i)}_{\text{new replica cost}} + \underbrace{\sum_{\substack{j=1, j \neq d, \\ X_{kj}(t_i)=1}}^{|DC|} (1 - X'_{kd}(t_i)) * X_{kd}(t_i) * c^r_{*kdj}(t_i)}_{\text{remove replica cost}} \Bigg)
\end{aligned}
\tag{1}
$$

For the proposed algorithm, the cost is composed of a direct cost associated with the total number of reads $r_{kd}$ and writes $w_{kd}$ executed for data $k$ in DC $d$ in the time interval. The direct cost factors $c^r_{kd}$ and $c^w_{kd}$ may vary for every DC and over time. To simplify the presentation, we assume that they remain constant during the respective time interval.

Next, there is a cost associated with local replicas due to updates that need to be applied to all DCs hosting replicas of the data item. The reads (cost factor $c^r_{kdj}$) and writes (cost factor $c^w_{kdj}$) that are directed from DC $d$ to $j$ if DC $d$ does not replicate the data item incur at least the cost for retrieving or sending the information to the closest DC $j$ holding the replica.

For the search cost, the expression $\rho_{kd}(t_i, t_{i+1})$ is 1 when there exists an operation between time $t_i$ to $t_{i+1}$, i.e. $r_{kd}(t_i, t_{i+1}) + w_{kd}(t_i, t_{i+1}) > 0$, or zero otherwise. For updates during this interval, information regarding the replica placement is needed if requests have to be forwarded. This corresponds to querying the placement information component.

Finally, $X'_{kd}$ denotes the new replication status after all the current operations have been executed, with some cost per notification of $c^r_{*kdj}$, and the cost of removing a replica from a DC of $c^r_{*kdj}$.

The total cost up to a time $T$ is then given by

$$Cost_k(T) = \sum_{i=0}^{n-1} Cost_k\left(t_i, t_{i+1}\right) \tag{2}$$

where the interval $(0, T)$ is split into subintervals $(t_i, t_{i+1})$, with $i \in [0, n]$, $t_i < t_{i+1}$, $t_0 = 0$, and $t_n = T$.

**Discussion** The model and basic protocol we derived for Adaptive Replication offers great flexibility as the influence of single terms can easily be prioritized by adapting the strength and cost factors. Further, it fits the SyncFree approach: Information about new replicas can be eventually propagated between the DCs, such that updates eventually reach every DC and can be applied to each replica. The deletion of replicas, however, requires stronger consistency and coordination if a minimum number of replicas must be maintained (e.g. to handle faults or network partitions, and to prevent data loss). [1] The placement information can be maintained in a dictionary service where all update operations on the dictionary are serialized. Queries on the dictionary do not need to be serialized. If a replica has been removed before the request reached the node, the origin node has to again query the dictionary. However, we expect replica removals to happen with low frequency. Also, as the strongly consistent operations are not on any critical path (i.e. blocking the highly frequent read and write operations to DC-local objects), they will incur only a small latency increase on average.

To further evaluate the applicability of the approach for distributed data stores, we implemented a prototype of our Adaptive Replication protocol in the Antidote platform. Further information regarding this implementation can be found in Deliverable D2.2.2, the accompanying report for the software deliverables.

[1] Similarly, strong consistency is also required for installing new replicas if a maximum number of replicas should not be exceeded.

## 4.5 Conflict-free Partially Replicated Data Structures

Depending on their usage, CRDTs such as Set CRDTs or Map CRDTs can grow quickly in size. Replicating large CRDTs in the clients can be a waste of resources, of both storage and bandwidth. For example, in a social-network application, the posts of a user wall can be stored in a Set CRDT. If the user is interested in only a small subset of these posts, storing the complete state of the CRDT at client-side is not necessary.

To address this issue, we have proposed a new abstraction, the Conflict-free Partially Replicated Data Structures (CPRDTs)[8]. A CPRDT is a CRDT that can be partitioned (or sharded) in multiple particles. We define particles as the smallest meaningful elements of a CPRDT, i.e. 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 inserted or looked up in the set. Specifications of CPRDTs require a definition of the particles that compose the CPRDT and the operations on these particles.

Each replica of a CPRDT $x_i$ maintains a set of particles, shard($x_i$). The replica only knows about the particles in shard($x_i$); therefore, it can only enable query and update operations that require and affect those particles. Furthermore, the CPRDT replica only needs to receive update operations that affect the particles in shard($x_i$) for convergence.

For each operation, the specification is extended with the following properties:

- **Affected particles** The function affected(*op*) returns the set of particles that may have their state affected after executing an update operation *op*.
- **Required particles** For an update or query operation *op*, required(*op*) denotes the set of particles needed by *op* to be properly executed. This means that, for replica $x_i$, an operation is enabled only if required(*op*) $\subseteq$ shard($x_i$). For example, for the lookup operation of a set, required(lookup($e$)) $= e$ where $e$ is an element of the set. In case $e \notin$ shard($x_i$), the replica will not be able to know whether $e$ is in the set.

These properties are needed in order to control the access to partially replicated CPRDTs. We have defined G-Set (grow-only set), OR-Set (observe-removed set) and a Tree CPRDT. The implementation of these CPRDTs has been evaluated on SwiftCloud as Antidote was not yet available at this time.

Extending the initial version of the paper that was included in Deliverable 3.1, we refined the causality and convergence criteria for CPRDTs. When constructing the causal history for a CPRDT, only the locally replicated shard set introduces causal dependencies. In particular, when merging CPRDT states of different replicas, only the intersection of the shards is affected by updates. Our initial model only touched on these topics, but missed a formal treatment.

## 4.6 CRDTs for partially incremental computations

Parallel to our work on CPRDTs, we developed an overview on computational CRDTs, that is, a class of CRDTs whose state is the result of a computation over the executed updates [29]. Most CRDTs proposed in literature are replicated forms

of collections. For such data types, a replica needs to maintain all data elements in all replicas. Thus, a model where every data replica maintains the same state and where all updates are propagated to all replicas is a natural fit.

In some cases, applications are not interested in actual elements or updates, but instead in the result of a computation over them. For this computational CRDTs, the state is the result of a computation over the executed updates. For example, a counter CRDT [32] counts the number of times an increment operation has been executed. In such cases, the replica does not need to maintain every individual update, but can instead maintain an integer that counts the number of increments executed at that replica. For synchronizing replicas, it suffices to propagate an integer that summarizes a set of updates.

We derived three generic designs that reduce the amount of information that each replica maintains and propagates for synchronizations for computational CRDTs. One of the designs deals with partially incremental computations.

**Example** An example of such an object is a top-K object for maintaining a high score list where an element can also be deleted. This use case has been adapted from an application in WP 1. Here, a value that does not belong to the top-K elements may later become part of the top, after a top element is deleted. To address this case, a possible approach is to use a Set CRDT to maintain the set of elements that have not been deleted. In this case, all replicas maintain the complete set, and all updates need to be propagated to all replicas. The top-K can be computed locally on the value of each replica.

In [29], we presented an alternative approach, in which each replica maintains all operations locally executed, and each replica only propagates to other replicas the operations that might affect the computed result. Each replica maintains a set of operations and the results of the computation performed at other sites — for simplicity of notation, we assume that the result of the computation is a subset of operations. An update operation updates the local set of operations. A read operation performs the computation considering the local operations and the results of the computation at the other replicas. For synchronizing replicas, a replica sends the results of its computations to all replicas and the subset of operations known locally that can affect the computed result at other replicas (in the top-K example, the delete of an element that belongs to the top elements). When receiving the state from a remote replica, the local replica is updated by merging the local set of operations with the remote operations that may affect the result of the computation, and by registering the most recent version of the computation for each site.

A top-K replicated data type that supports add $\big(n, v\big)$ and del $\big(n\big)$ operations can be defined as follows:

$$V_0 = \{\}$$
$$\text{fun}(s) = \text{maxk}\left(\{o \in s : o = \text{add}(n, v) \wedge (\nexists o' \in s : o \prec o' \wedge o' = \text{del}(n))\}\right)$$

where $V_0$ denotes the initial state, fun the computation associated with the application of an update, and maxk($s$) a function that returns the $k$ add $\big(n, v\big)$ operations with largest values for $v$.

This design enforces eventual consistency, assuming that replicas continue synchronizing until they reach an equivalent state, i.e., a state where read operations

return the same result in every replica. However, this may not happen after the first synchronization step. For example, consider a top-1 object replicated in two sites: Site 1 executed operations $\{\text{add}(b, 15), \text{add}(a, 10)\}$ and site 2 executed operations $\{\text{add}(b, 16), \text{add}(c, 12)\}$. The two sites synchronize, with the top-1 element, $(b, 16)$, being known at both replicas. Later, $\text{del}(b)$ executes at site 1, promoting $(a, 10)$ to the top at site 1. After the propagation of $\text{del}(b)$ to site 2, $(c, 12)$ is promoted to the top at site 2. Only after the next synchronization step, the top at site 1 $(a, 10)$ is replaced by the same value as in site 2 $(c, 12)$.

Our work on computational CRDTs was focused so far on their classification and theoretical properties. Implementations for Counter CRDTs already exist in Antidote and Riak. Other computational CRDTs have been specified (in collaboration with WP1 and WP3), but not yet integrated into the platforms. Further examples for computational CRDTs are currently being developed in WP4 and will be presented in the next deliverable for WP4.

## 4.7 Final remarks

**Swiftcloud** The task related to this deliverable says that we want to investigate how "to support CRDTs in a large number of nodes with relatively stable connectivity" on a "base platform with numerous nodes located near clients". This setting has been intensively studied in the context of the Swiftcloud platform [39]. The Swiftcloud platform and associated protocols have been described already in Deliverable 2.1., where we focused on the server side aspects. Here, we only want to highlight the partial replication aspects in the Swiftcloud platform (for further detail see Appendix F with the most recent paper draft).

SwiftCloud ensures causally consistent, available, and convergent access to the cloud database from client nodes. A flexible client-server topology both enables small meta-data and ensures fault-tolerance.

Swiftcloud employs two types of replication strategies: Between DCs, full replication is used to propagate updates; for clients, only the data items requested ( recently) by a client are replicated at client-side. To this end, SwiftCloud uses novel meta-data decoupling

- *tracking causality* with small vectors, sized in the number of DCs, referring to DC-assigned timestamps, from
- *unique identification of an update* with client-assigned timestamps, which protect from duplicated update execution.

Thanks to funneling updates through DCs, the size of meta-data remains small and stable, at the expense of staleness, but without affecting correctness.

**Causal dependency under fail-over** When a client switches to a different DC, the state of the new DC may be unsafe, because some of the client 's causal dependencies are missing. Some geo-replication systems avoid creating dangling causal dependencies by making synchronous writes to multiple data centres, at the cost of high update latency [9]. Others remain asynchronous or rely on a single DC, but after failover clients are either blocked or they violate causal consistency [22, 24, 21].

The former systems trade consistency for latency, the latter trade latency for consistency or availability.

An alternative approach would be to store the dependencies on the client . However, since causal dependencies are transitive, this might include a large part of the causal history and a substantial part of the database.

Our approach is to make clients co-responsible for the recovery of missing session causal dependencies at the new DC. SwiftCloud lets a client observe a remote update only if it is stored in a number $K > 1$ of DCs. Such *$K$-stable* versions are likely to be in other DCs. This does not harm consistency, because the client observes his own earlier updates from the local replica.

Our protocols let a client observe only the union of:

- its own updates, in order to ensure the "read-your-writes" session guarantee [33], and
- the $K$-durable updates made by other client s, to ensure other session guarantees, hence causal consistency.

In other words, the client depends only on updates that the client itself can send to the new DC, or on ones that are likely to be found in a new DC. When failing over to a new DC, the client helps out by checking whether the new DC has received its recent updates, and if not, by repeating the commit protocol with the new DC. SwiftCloud prefers to serve a slightly old but $K$-durable version, instead of a more recent but more risky version. Instead of the consistency and availability vs. latency trade-off of previous systems, SwiftCloud trades availability for staleness.

The work on Swiftcloud and Charcoal are complementary. It remains to show whether the genuine partial replication protocol of Charcoal can be used for the server-side replication in Swiftcloud.

**Antidote** In addition to the work on partial replication, there has been on-going work on the Antidote platform. Recent changes include support for benchmarking and the setup of a continuous integration server to facilitate development (in collaboration with WP5). In addition to the extended ClockSI protocol that we presented in Deliverable 2.1, we now also support other protocols such as Eiger [26], COPS [27] and GentleRain [18]. Antidote now also features bounded counters [6] that have been developed in WP3 and described in Deliverable D3.1. Further, we support now a protocol buffer interface which simplifies the adaptation of applications written for the Riak key-value store to the Antidote platform.

More details regarding the Antidote platform and the other software deliverables can be found in the report accompanying Deliverable 2.2.2.

**CRDT Composition** The task description says that this deliverable deals also with CRDT composition under partial replication. Both Antidote and Charcoal provide transactions on CRDTs and hence a typical form of CRDT composition. As the transactional interface and its semantics have been already covered in Deliverable 2.1.1 and Deliverable 3.1. from WP 3, we do not cover them in this report. Other forms of composition, such as Map CRDTs or general composition techniques using inflations, have also been presented in detail in Deliverable 3.1.

# 5 Papers and publications

The work performed in WP2 and in collaboration with other work packages has led to several publications.

The following papers and technical reports relate to this deliverable:

- [35] Alejandro Zlatko Tomsic, Tyler Crain, and Marc Shapiro. An empirical perspective on causal consistency. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM
- [11] Tyler Crain and Marc Shapiro. Designing a causally consistent protocol for geo-distributed partial replication. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM
- [10] Tyler Crain and Marc Shapiro. Charcoal: A causally consistent protocol for geo-distributed partial replication. Technical report, INRIA, mar 2015
- [4] Amadeo Ascó and Annette Bieniusa. Adaptive strength geo-replication strategy. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM
- [29] David Navalho, Sérgio Duarte, and Nuno Preguiça. A study of crdts that do computations. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM

The following paper is under submission:

- [8] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. Under submission
- [38]Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. Write fast, read in the past: Causal consistency for client-side application. Under submission

All papers named here are attached to this report.

# References

[1] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[2] Peter M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems*, 13:263–304, 1988.

[3] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. G-DUR: A middleware for assembling, analyzing, and improving transactional protocols. In *Int. Conf. on Middleware (MIDDLEWARE)*, pages 13–24, Bordeaux, France, December 2014.

[4] Amadeo Ascó and Annette Bieniusa. Adaptive strength geo-replication strategy. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM.

[5] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 761–772, New York, NY, USA, 2013.

[6] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno M. Preguiça.

[7] N Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pages 59–72, San Jose, CA, USA, May 2006. Usenix, Usenix.

[8] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. Under submission.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In Thekkath and Vahdat [34], pages 261–264.

[10] Tyler Crain and Marc Shapiro. Charcoal: A causally consistent protocol for geo-distributed partial replication. Technical report, INRIA, mar 2015.

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

[12] Xiaohua Dong, Ji Li, Zhongfu Wu, Dacheng Zhang, and Jie Xu. On dynamic replication strategies in data service grids. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 155–161, May 2008.

[13] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[14] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[15] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pages 11:1–11:14, Santa Clara, CA, USA, October 2013. Assoc. for Computing Machinery.

[16] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 173–184, Braga, Portugal, October 2013. IEEE Comp. Society.

[17] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Closing the performance gap between causal consistency and eventual consistency,. In *W. on the Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, the Netherlands, 2014.

[18] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke, editors, *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014*, pages 1–13. ACM, 2014.

[19] Avinash Lakshman and Prashant Malik. Cassandra, a decentralized structured storage system. In *W. on Large-Scale Dist. Sys. and Middleware (LADIS)*, volume 44 of *Operating Systems Review*, pages 35–40, Big Sky, MT, USA, October 2009. ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery.

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

[21] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Thekkath and Vahdat [34], pages 265–278.

[22] 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.

[23] 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 *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, October 2011. Assoc. for Computing Machinery.

[24] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 313–328. USENIX Association, 2013.

[25] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, pages 313–328, 2013.

[26] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 313–328, Lombard, IL, USA, April 2013.

[27] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual consistency. *Commun. ACM*, 57(5):61–68, May 2014.

[28] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

[29] David Navalho, Sérgio Duarte, and Nuno Preguiça. A study of crdts that do computations. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM.

[30] João Paiva and Luís Rodrigues. On data placement in distributed systems. *Operating Systems Review*, 49(1):126–130, 2015.

[31] Masoud Saeida Ardekani, Pierre Sutra, Marc Shapiro, and Nuno Preguia. On the scalability of snapshot isolation. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 369–381. Springer Berlin Heidelberg, 2013.

[32] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[33] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, September 28-30, 1994*, pages 140–149. IEEE Computer Society, 1994.

[34] Chandu Thekkath and Amin Vahdat, editors. *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 2012.

[35] Alejandro Zlatko Tomsic, Tyler Crain, and Marc Shapiro. An empirical perspective on causal consistency. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM.

[36] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, June 1997.

[37] Ouri Wolfson and Amir Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, March 1991.

[38] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. Write fast, read in the past: Causal consistency for client-side application. Under submission.

[39] Marek Zawirski, Annette Bieniusa, Valter Balegas, Sérgio Duarte, Carlos Baquero, Marc Shapiro, and Nuno Preguiça. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. *arXiv preprint arXiv:1310.3107*, 2013.

# A Designing a causally consistent protocol for geo-distributed partial replication

# Designing a causally consistent protocol for geo-distributed partial replication

Tyler Crain
Inria Paris-Rocquencourt &
Sorbonne Universités, UPMC Univ Paris 06, LIP6
tyler.crain@lip6.fr

Marc Shapiro
Inria Paris-Rocquencourt &
Sorbonne Universités, UPMC Univ Paris 06, LIP6
marc.shapiro@acm.org

## Abstract

Modern internet applications require scalability to millions of clients, response times in the tens of milliseconds, and availability in the presence of partitions, hardware faults and even disasters. To obtain these requirements, applications are usually geo-replicated across several data centres (DCs) spread throughout the world, providing clients with fast access to nearby DCs and fault-tolerance in case of a DC outage. Using multiple replicas also has disadvantages, not only does this incur extra storage, bandwidth and hardware costs, but programming these systems becomes more difficult.

To address the additional hardware costs, data is often *partially replicated*, meaning that only certain DCs will keep a copy of certain data, for example in a key-value store it may only store values corresponding to a portion of the keys. Additionally, to address the issue of programming these systems, consistency protocols are run on top ensuring different guarantees for the data, but as shown by the CAP theorem, strong consistency, availability, and partition tolerance cannot be ensured at the same time. For many applications availability is paramount, thus strong consistency is exchanged for weaker consistencies allowing concurrent writes like *causal consistency*. Unfortunately these protocols are not designed with partial replication in mind and either end up not supporting it or do so in an inefficient manner. In this work we will look at why this happens and propose a protocol designed to support partial replication under causal consistency more efficiently.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 609551.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*PaPoC'15*, April 21, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2745947.2745953

## 1. Partial replication

Partial replication is becoming essential in geo-replicated systems to avoid spending uncescessary resources on storage and networking hardware. Implementing partial replication is more difficult than deciding how many replicas to have because protocols for data consistency must hide the organisation of replicas so that the programmer sees the data as a single continuous store. Furthermore ensuring consistency with partial replication does not always easily scale, as it often requires additional communication between nodes not involved in the operations. For example, in [8], Saeida shows that a scalable implementation of partial replication, namely one that ensures genuine partial replication [9] is not compatible with the snapshot-isolation consistency criterion. Differently, this work focuses on *causal consistency* which allows concurrent writes and uses meta-data propagation instead of synchronisation to ensure consistency, but even in this case, implementing partial replication in a scalable way is not a straightforward.

### 1.1 Causal consistency and partal replication

While protocols ensuring causal consistency are generally efficient when compared to strongly consistent ones, they often do not support partial replication by default or if they do, limit scalability by requiring coordination with nodes that do not replicate the values updated during propagation. Within the standard structure of these protocols, updates are performed locally, then propagated to all other replicas where they are applied respecting their *causal order*, which is given by session order and reads-from order or can be defined explicitly. Given the asynchrony of the system, propagated updates might arrive out of causal order at external replicas, thus before they are applied a *dependency check* must be performed to to ensure the correctness. This check is based on ordering meta-data that is propagated along with the updates or through separate messages [2].

The best known structure of this meta-data are vector-clocks where each totally-ordered participant is given a vector entry and each of their updates are assigned a unique increasing scalar value. Since these geo-replicated systems can have a large number of participants, they often use slightly different representations of the vector-clocks. For example, certain protocols use vectors with one entry per DC [13], or one entry per partition of keys [3], or use vectors that can be trimmed based on update stability or the organisation of the partitions of keys across DCs [3]. Other than vector clocks other approaches exist including real time clocks [5], or to track reads of memory locations [6] or operations [7] up to the the last update performed by this client.

Interestingly, all of these mechanisms create (over) approximations of the dependencies of each operation, i.e. from this meta-data you cannot tell exactly what the causal dependencies for this operation are, but for correctness they cover at minimum all the dependencies. For example when using a single vector entry per server, all operations from separate clients connected to this server will be totally ordered even if they access disjoint sets of data.

Such systems use these approximations primarily because precise tracking of dependencies would not scale as the size of the meta-data would grow up to the order of the number of objects or users in the systems (depending on how dependencies are tracked). The issue with over approximating dependencies is that the dependency check might have wait on more dependencies than necessary, allowing the client to read stale versions of the data. Fortunately though this does not block the progress of clients as updates are replicated outside of the critical path.

### 1.2 Partial replication and approximate dependencies

This approximate tracking of dependencies also creates an unintended effect of making partial replication more costly.

To see why this happens, consider the example shown in figure 1 where a protocol is used that tracks dependencies using a version vector with an entry per server. Assume this protocol supports partial replication by only sending updates and meta-data to the servers that replicate the concerned object. In this example there are two DCs $DC_A$ and $DC_B$ each with 2 servers: $A_1$ and $A_2$ at $DC_A$ and $B_1$ and $B_2$ at $DC_B$. Server $A_1$ replicates objects $x_1$ and $x_2$, server $B_1$ replicates objects $x_2$ and $x_3$ while server $A_2$ replicates objects $y_1$ and $y_2$ and server $B_2$ replicates objects $y_2$ and $y_3$. The system starts in an initial state where no updates have performed. Consider then that a client performs an update $u1$ on object $x_1$ at server $A_1$, resulting in the client having a dependency vector of $[1, 0, 0, 0]$, with the 1 in the first entry of the vector representing the update $u1$ at $A_1$. Since $x_1$ is not replicated elsewhere the update stays locally at $A_1$. Following this, the client performs an update $u2$ on object $y_2$at server $A_2$, returning a dependency vector of $[1, 1, 0, 0]$. The update $u2$ is then propagated asynchronously to $B_2$, where upon arrival a dependency check is performed. Since the dependency vector inclues a dependency from $A_1$, before applying the update, $B_2$ must check with $B_1$ that it has received any updates covered by this dependency in case they were on a key replicated by $B_1$. But since $B_1$ has not heard from $A_1$ it does not know if the update was delayed in the network, or if the update involved an object it does not replicate. Thus $B_1$ must send a request to $A_1$ checking that is has received the necessary update. $A_1$ will then reply that it is safe because $u1$ did not modify an object replicated by $B_1$, which will then be forwarded to to $B_2$ at which time $u2$ can be safely applied. Notice that if dependencies were tracked precisely, this additional round of dependency checks would not be necessary as the dependency included with $u2$ would let the server know that it only depended on keys not replicated at $DC_B$.

While this is a simple example that one could imagine easily fixing, different workloads and topologies can create complex graphs of dependencies that are not so easily avoided. Furthermore, current protocols designed for full replication do not take any additional measures specifically to minimize this cost, instead they suggest to send the meta-data to every DC as if it was fully replicated either in a separate channel [2] or simply without the update payload. In effect using no specific design patterns to take advantage of partial replication.

## 2. An initial approach

The goal of this work then is to develop a protocol supporting partial replication and providing performance equal to fully replicated protocols in a full replication setting, while minimising dependency meta-data and checks in a partial replication setting. We will now give a short description of the main mechanisms used to design this algorithm. It should be noted that these mechanisms are common to many protocols supporting causal consistency, except here they are combined in a way with the goal of supporting partial replication.

- **Update identification** Vector clocks are the most common way to support causality. To avoid linear growth of vector clocks in the number of (client) replicas, we apply a similar technique as in the work of Zawirski et al.[13]. Each entry in a vector represents a DC, or more precisely a cluster within a DC. Modified versions of protocols such as ClockSI [4] or a DC- local service handing out logical timestamps, such as a version counter, can be used to induce a total order to the updates issued at this DC, which can then be represented in the DC's vector entry.

  For causality tracking, each update is associated with its unique timestamp given by its home DC, plus a vector clock describing its dependencies. To provide session guarantees such as causally consistent reads when interacting with clients, the client keeps a vector reflecting its

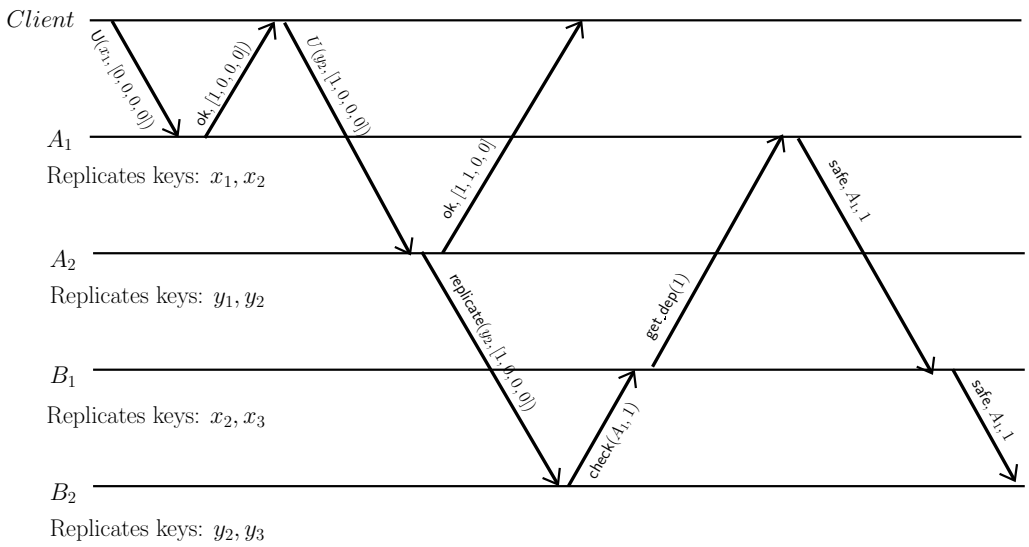



Figure 1: An example showing the possible dependency checks needed to ensure causality in a system with partial replication using version vectors with one entry per server for tracking dependencies.

previously observed values and writes. The system then ensures that clients may only read values containing all dependencies given this vector.

Given that in partial replication a DC might not replicate all objects, certain reads will have to be forwarded to other DCs where the object being read is replicated. The receiving DC then uses the client's dependency vector to generate a consistent version of the object that is then forwarded to be cached at the DC the client is connected to.

- **Disjoint safe-time metadata** In general, most protocols ensure causal consistency by not making updates from external DCs visible locally to clients until all updates causally preceding it have been received. When objects are replicated at all DCs this is fairly straightforward as all dependent updates are expected to be received. This is not always the case in partial replication since only the replicated dependent operations should be received, which could result in additional messages or dependency checks (see figure 1 for an example of why these additional checks would be needed), something which we are trying to avoid in order to have an efficient implementation.

  To avoid these additional dependency checks and metadata, the key insight in this work is to perform the dependency calculation at the origin DC and not the receiving DCs. Updates are still sent directly to their sibling replicas at other DCs, but they are not made visible to readers at the receiving DC until the origin DC confirms that its dependencies have been received i.e. the origin DC tracks which of its updates are safe to make visible at the receiver. At the origin DC, updates issued up to a time $t$ are considered safe to apply at a receiving DC when all of the origin DC's servers have sent all their updates on the replicated objects of the receiver items up to time $t$. To keep track of this, a server at the origin DC communicates with each local server, keeping track of the time of the latest updates sent to external DCs, and once it has heard from each local server that time $t$ is safe, this information is then propagated to the external DCs as a single message. Doing this avoids unnecessary cross-DC dependency checks and meta-data propagation, saving computation and network bandwidth. The negative consequence of this is that the observable data at the receiving DC might be slightly more stale than in the full replication case because the receiving DC has to wait until the sending DC has let it know that this data is safe. Such a delay can be seen as a consequence of tracking dependencies approximately as seen in the example in figure 1

- **Local writes to non-replicated keys** Given that causal consistency allows for concurrent writes, in order to ensue low latency and high availability a DC will accept writes for all objects, including those that it does not replicate. Using the vector clocks and metadata as described above this can be done without any additional synchronisation by just assigning unique timestamps to these updates that are reflected in the vector of the local DC. These updates can then be safely logged and made durable even in the case of network partition.

- **Atomic writes and snapshot reads** Beyond simple key-value operations, the protocol provides a weak form of transactions which allows to group reads and updates together and supporting CRDT objects [10]. Atomic writes

can be performed at the local DC using a 2-phase commit mechanism without contacting the remote replicas in order to allow for low latency and high availability. The updates are then propagated to the other DCs using the total ordered dependency metadata described previously ensuring their atomicity. (Note that atomic updates can include keys not replicated at the origin DC.) Causally consistent snapshot reads can be performed at a local DC by reading values according to a consistent vector clock, where reads of data items not replicated at the local DC are performed at another DC using the same vector clock.

Using these mechanisms allow partial or full replication with causal consistency while limiting the amount of unnecessary inter-DC meta-data traffic. All DCs are able to accept writes to any key, and causally consistent values can be read as long as one replica is available. Additionally the way the keys are partitions within a DC is transparent to external DCs, allowing this to be maintained locally.

An implementation of this protocol [12] is being developed within Antidote [11], the research platform for the SyncFree FP7 project, which is built on top of Riak-core [1] designed for testing scalable geo-replicated protocols.

Finally, it is important to note that while this protocol helps mitigate some of the costs of implementing partial replication in previous protocols, it does not completely solve the problem. It still uses imprecise representation of dependencies, which can still lead to false dependencies (that are checked locally within the sending DC) and can result in reading stale values that might otherwise be safe to read. Further studies are still needed and novel approaches using different mechanisms to see if these costs can be avoided entirely.

## References


[1] Basho. Riak-core. `https://github.com/basho/riak_core`, 2015.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pages 59–72, San Jose, CA, USA, May 2006. Usenix, Usenix. URL `https://www.usenix.org/legacy/event/nsdi06/tech/belaramani.html`.

[3] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pages 11:1–11:14, Santa Clara, CA, USA, Oct. 2013. Assoc. for Computing Machinery. . URL `http://doi.acm.org/10.1145/2523616.2523628`.

[4] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 173–184, Braga, Portugal, Oct. 2013. IEEE Comp. Society. . URL `http://doi.ieeecomputersociety.org/10.1109/SRDS.2013.26`.

[5] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Closing the performance gap between causal consistency and eventual consistency,. In *W. on the Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, the Netherlands, 2014. URL `http://eventos.fct.unl.pt/papec/pages/program`.

[6] 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 *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, Oct. 2011. Assoc. for Computing Machinery. .

[7] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages 313–328, Lombard, IL, USA, Apr. 2013. URL `https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf`.

[8] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguia. On the scalability of snapshot isolation. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 369–381. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40046-9. . URL `http://dx.doi.org/10.1007/978-3-642-40047-6_39`.

[9] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 214–224, New Dehli, India, Oct. 2010. IEEE Comp. Society. URL `http://doi.ieeecomputersociety.org/10.1109/SRDS.2010.32`.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. 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, Oct. 2011. Springer-Verlag. . URL `http://www.springerlink.com/content/3rg3912287330370/`.

[11] SyncFree. Antidote reference platform. `https://github.com/SyncFree/antidote`, 2015.

[12] SyncFree. Antidote reference platform - partial replication branch. `https://github.com/SyncFree/antidote/tree/partial_replication`, 2015.

[13] 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. *arXiv preprint arXiv:1310.3107*, 2013.

# B An empirical perspective on causal consistency

# An empirical perspective on causal consistency

Alejandro Z. Tomsic
Inria Paris-Rocquencourt &
Sorbonne Universités, UPMC Univ Paris 06, LIP6
alejandro.tomsic@lip6.fr

Tyler Crain Marc Shapiro
Inria Paris-Rocquencourt &
Sorbonne Universités, UPMC Univ Paris 06, LIP6
tyler.crain@lip6.fr/marc.shapiro@acm.org

## Abstract

Causal consistency is the strongest consistency model under which low-latency and high-availability can be achieved. In the past few years, many causally consistent storage systems have been developed. The long-term goal of this initial work is to perform a deep study and comparison of the different implementations of causal consistency. We identify that protocols that provide causal consistency share the well-known DUR (deferred update replication) algorithmic structure and observe that existing implementations of causal consistency fall into a sub-category of DUR that we name *A-DUR (Asynchronous-DUR)*. In this work, we present the A-DUR algorithmic structure, the pseudocode for the instantiation of two causally consistent protocols under the G-DUR framework, and describe the empirical study we intend to perform on causal consistency.

## 1. Introduction

The CAP theorem [7] proves that no distributed service can provide strong consistency, availability and partition-tolerance simultaneously; one must be sacrificed. In a distributed setting, network partitions are a given. As a consequence, in the past years, there has been a big amount of research destined to understand the tradeoffs between consistency and availability. Causal consistency has proven to be in the sweetest spot of this tradeoff, i.e., it is the strongest consistency model under which low-latency and high-availability can be achieved [13]. This model is easier to reason about for programmers than eventual consistency, its previously widely-adopted weaker counterpart.

In the past few years, many causally consistent systems have been developed [4–6, 11, 12]. These systems differ in their implementation due to the assumptions and compromises they make. For instance, there are protocols that track potential dependencies [5, 6, 11, 12]; defined by the happens-before relation [10] between events, while others just track explicit dependencies [4, 9]. Another important trade-off is visibility latency vs. throughput [3, 6]. In this line, most protocols use explicit dependency check messages; which improves visibility [1, 4–6, 11, 12] while others improve throughput by utilising a stabilisation mechanism [6] that slightly penalises it.

Choosing among a large number of systems that provide causal consistency can be hard. Even when protocols are well documented, the used vocabulary, naming conventions and perspectives vary. Moreover, design considerations, topology assumptions and implementation differences further constrain the possibility of a fair comparison. Finally, most protocols only compare to a few alternatives and/or to an eventually- or strongly-consistent baseline. It thus remains complicated to understand the important differences among causally-consistent protocols and to make an objective, scientific comparison of their behaviour.

The long-term goal of this initial work is to perform a deep comparative study of the different implementations of causal consistency. As a first-step towards that goal, we need an environment where to implement different protocols. In recent work, Saeida Ardekani et al. [2] identified that there is a family of strongly-consistent protocols that share a generic algorithmic structure, called DUR (deferred update replication). Briefly, DUR protocols execute transactions in two phases: an execution phase, were values are read and updates are buffered; and a termination phase, where an atomic commitment protocol decides on committing or aborting the transaction, and its effects are propagated across the system. Their work presented the G-DUR framework, a tool for implementing DUR protocols, and a deep empirical comparison of relevant strongly consistent systems.

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 609551.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*PaPoC'15*, April 21, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2745947.2745949

We identify that causally-consistent protocols also present a DUR structure. Furthermore, we observe that most implementations of causal consistency fall into a sub-category of DUR that we name *A-DUR (Asynchronous-DUR)* and explain in Section 2.

Based on the aforestated, we have decided to implement the protocols for performing our study under G-DUR[2]. In this work, we present the pseudocode for the G-DUR instantiation of two causally consistent protocols[6, 12] (Section 2.1) and describe the empirical study we intend to perform on causal consistency (Section 3).

## 2. The A-DUR property

A-DUR protocols present particular properties that are not present in every DUR protocol: $(i)$they are topology (data center) aware; $(ii)$transactions execute and commit locally: communication only involves local replica(s) of each updated object (normally, the one(s) located at the DC where the transaction is started); $(iii)$they only incur a termination phase in the case of atomic writes, which never aborts a transaction, and, most distinctively; $(iv)$they perform *background asynchronous processing*, which handles tasks like propagating committed updates to remote DCs; checking dependencies, resolving conflicts (i.e., causal+ convergence) and applying updates at remote DCs; and/or making updates visible.

### 2.1 A-DUR protocols under G-DUR

In this section, we present the G-DUR instantiation of two systems that provide causal consistency: GentleRain and Eiger. Namely, the G-DUR instantiation entitles defining the implementations of the following functions: a function $\theta$ for partially ordering transactions; $choose$, for choosing a consistent value when reading an object; $certifying\_obj(T_i)$, for determining which objects will be certified during termination phase of a transaction; $commute(T_i, T_j)$, for determining if two transactions $T_i$ and $T_j$ commute; $certify(T_i)$, for deciding if a transaction is safe to be committed; $async\_proc$, for describing the asynchronous processing the protocol performs; and $dep\_check(T_i)$, for checking the dependencies of a transaction $T_i$ at a remote DC. Note that the last two functions are defined specifically for A-DUR instantiations and replace the $post\_commit$ function present in G-DUR. The rest of the code needed to fully implement these protocols is given by the general G-DUR structure [2].

***Eiger*** Eiger[12] (see Algorithm 1) tracks potential dependencies and uses explicit dependency check messages to decide when updates are made visible. Each version of an object stores an identifier composed by a logical timestamp plus a server identifier; and a set of one-hop dependencies which determine causal ordering (line 1). In order to ensure causality, a read operation selects the latest version of an object (Eiger's one round read transaction). When the latest version is not suitable for establishing a causally-consistent snapshot, the read protocol incurs in a second round of reads that selects a version using a timestamp provided by the transaction coordinator (line 2). In order to provide atomic write operations, this protocol relies on a two-phase commit with positive cohorts and indirection (explained elsewhere[12]) that never aborts and only involves the replicas at the local DC (2PC-PCI$_{local}$) holding an object written by the transaction (lines 3-6). After a transaction commits, its effects are sent in the background to the replicas of the updated objects in remote DCs (line 7). At a receiving DC, each server hosting an objected updated by $T_i$ performs a dependency check. It sends messages to the servers at its datacenter in order to check that they have applied the operations identified in each updated object's metadata (line 9).

**Algorithm 1** G-DUR instantiation of Eiger

1: $\theta \equiv \{TS\}$
2: $choose \equiv choose_{last\_\{TS\}} \vee choose_{cons}$
3: $\mathcal{AC} \equiv$ 2PC-PCI$_{local}$
4: $certifying\_obj(T_i) \equiv ws(T_i)$
5: $commute(T_i, T_j) \equiv true$
6: $certify(T_i) \equiv true$
7: $async\_proc \equiv send(ws(T_i) \cup \theta(T_i))$
8: to $replicas(ws(T_i)) \setminus local\_replicas(ws(T_i))$
9: $dep\_check(T_i) \equiv \forall\, ts_i \in \theta(T_i)$ :
10: $applied_{local}(op_{ts_i.id})$

***GentleRain*** GentleRain[6] (see Algorithm 2) tracks potential dependencies and uses a global stabilisation protocol to make object versions visible. Each version of an object is identified by a physical timestamp, which is used to determine causally consistent snapshots (lines 1 and 2). This protocol does not incur in a termination phase as it does not provide atomic writes. After an object is updated locally, the update is sent in the background to the replicas of the updated objects in remote DCs (line 5). Each server periodically exchanges its local physical clock ($\theta(p)$)with the rest of the servers in the system to compute the GST (global stable time) that is used to make updates visible.

**Algorithm 2** G-DUR instantiation of GentleRain

1: $\theta \equiv TS$
2: $choose \equiv choose_{cons}$
3: $certify(T_i) \equiv true$
4: $async\_proc_1 \equiv send(\theta(p))$ to $\Pi$
5: $async\_proc_2 \equiv send(ws(T_i) \cup \theta(T_i))$
6: to $replicas(ws(T_i)) \setminus local\_replicas(ws(T_i))$
7: $dep\_check(T_i) \equiv GST < \theta(T_i)$

## 3. Causal Consistency Study

In this section, we briefly describe the study we intend to realise.

We will start by implementing causally consistent protocols on top of the G-DUR framework, which we expect to save us a significant amount of coding effort. As part of this step, we plan to modify the framework itself in order to optimally support the A-DUR structure. Our study will focus (not exclusively) on a comparison of the realised protocols, an analysis of their bottlenecks and an assessment on the processing and communication costs of causal consistency. In particular, we will analyse the performance of protocols by considering the following tradeoffs and design considerations: potential vs. explicit causality tracking; dependency check messages vs. global stabilisation protocols, provided transactional API, metadata overhead, topology assumptions (partitioning scheme, partial vs. full replication), inter DC replication mechanisms, data staleness and type of causal+ convergence provided (or lack thereof). We plan to run our experiments in Grid'5000 [8], under different configurations and workloads.

With the results obtained from this study, we plan to identify the different flavours of causal consistency and to conclude on the costs of implementing them, when compared to strong and weak consistency.

## References


[1] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on Chain Replication. Apr. 2013.

[2] M. S. Ardekani, P. Sutra, and M. Shapiro. G-dur: A middleware for assembling, analyzing, and improving transactional protocols. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 13–24, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2785-5. . URL http://doi.acm.org/10.1145/2663165.2663336.

[3] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. 2012.

[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. ISBN 978-1-4503-2037-5. . URL http://doi.acm.org/10.1145/2463676.2465279.

[5] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. pages 11:1–11:14, Santa Clara, CA, USA, Oct. 2013. . URL http://doi.acm.org/10.1145/2523616.2523628.

[6] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3252-1. . URL http://doi.acm.org/10.1145/2670979.2670983.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700. .

[8] Grid'5000. Grid'5000, a scientific instrument [...]. https://www.grid5000.fr/, retrieved April 2013.

[9] 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. URL http://dx.doi.org/10.1145/138873.138877.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. URL http://doi.acm.org/10.1145/359545.359563.

[11] 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. pages 401–416, Cascais, Portugal, Oct. 2011. .

[12] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. pages 313–328, Lombard, IL, USA, Apr. 2013. URL https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf.

[13] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

# C Adaptive Strength Geo-Replication Strategy

# Adaptive Strength Geo–Replication Strategy

Amadeo Ascó Signes
Trifork Leeds
Leeds, UK
aas@trifork.com

Annette Bieniusa
Technische Universität Kaiserslautern
Kaiserslautern, Germany
bieniusa@cs.uni-kl.de

## Abstract

The amount of data being processed in Data Centres (DCs) keeps growing at an enormous rate so that full replication may start being impractical. The application of replication between DCs is used to increase data availability in the presence of site failures and to reduce latency by accessing the data closely located, if possible. This means that replicating the data only in some of the DCs is becoming more critical in order to reduce the costs of keeping the data (weakly) consistent while maintaining high availability (scalability) and low access costs. When data read and write request patterns change, then deciding which data should be replicated and where needs to be made dynamically. Given that the problem of finding an optimal replication schema in a general network has been shown to be NP-complete for the static case, so it is unlikely that there exists a general algorithm for an optimal solution to the dynamic problem.

We present here a new adaptive bio–inspired replication strategy, which is completely decentralised, adaptive, and event-driven, inspired on the Ant Colony algorithm.

***Keywords*** Adaptive Replication, Geo–replication, Large–Scale Database Replication, Accessibility

## 1. Introduction

The amount of data being processed in DCs keeps growing at enormous rate [5, 6, 16]. Some of the areas where the amount of stored data already reach terabytes (TBs) and even petabytess (PBs) are data mining, particle physics, climate modelling, high energy physics and astrophysics, to name a few, all including data which needs to be shared and analysed [10, 13, 14]. DCs are able to ensure that stored data is highly accessible and scalable. But the location of a DC in respect of the client accessing the data has an impact on availability, access times (latency – accessibility) and costs derived from providing the data. Replicating subsets of the data at multiple sites is a possible solution to reduce some of these undesirable effects, [1, 4, 17]. An increase in the number of replicas may result in large bandwidth savings and lead to a reduction in user response time on reads or writes depending on the replication type, i.e. replication on read or write. But keeping too many replicas of the data incurs an extra cost, such as network traffic to keep all versions of the data consistent, additional storage and computational power [9, 15].

This means that finding an optimal replication distribution that minimises the amount of network traffic given certain read and write patterns for various objects should alleviate these extra costs when replicating [4, 11, 19]. Given the volume of operations considered and the expected access speed, any algorithm suitable to be applied to this constraint optimisation problem must have a very fast execution time or it will have a detrimental effect on the latency.

## 2. Classification of Strategies

Replication strategies can be grouped, based on their variability, into two types: **static replication** where a replica persists until it is deleted by a user or its duration expires, and **dynamic replication** where the creation and deletion of replicas are managed automatically and normally directed by the users access pattern to the data [7]. In static replication, the major drawback is the inability to adapt to changes in the data access pattern.

Under **partial replication** a data item can be decomposed into several parts, which may be located in different places in the overall system, i.e. DCs, within a DC in different nodes or at the client-side [4, 4, 15], whereas the focus of **adaptive geo–replication** is to decide what data, where the data is located within the overall system of DCs and how many replicas exist simultaneously, [1, 2, 10, 12, 18]. Both of these types maybe be combined to obtain further performance improvements.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*PaPoC'15*, April 21, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04. . . $15.00.
http://dx.doi.org/10.1145/2745947.2745950Reprinted from PaPoC'15, [Unknown Proceedings], April 21, 2015, Bordeaux, France, pp. 1–4.

## 3. Adaptive Strength Geo–replication Strategies

The problem of finding an optimal geo–replication schema in a general network has been shown to be NP-complete for the static case [3, 20, 21]. Hence, there is no known efficient geo–replication algorithm for locating a convergent optimal solution dynamically. Given this, we propose an algorithm based on Wolfson's adaptive algorithm [19] for replicated data in a distributed system. The algorithm takes into account the changes in the read-write pattern of the processors in the network. It is based on the principles of the Ant Colony Optimisation algorithms, which are inspired by the behaviour of ant colonies when deciding which path to follow when foraging [8].

It should be noted that we consider here the main purpose of replication not to be the recovery from disasters as this should be the responsibility of special recovery data centres. Neither it is the DC responsibility to provide analytical services as these are provided by the data warehouse(s). The main propose of the considered DCs is to provide operational access to clients (operational DCs), which corresponds to intensive read/write operations to the client's most recent data.

The general idea of the algorithm we propose is to decide without the need of human intervention where and when to replicate with the main objective of reducing the latency and network traffic (reduce usage of bandwidth).

In general terms, any read operation in a DC reinforces the need to have a replica of the data in this DC. Similarly, but perhaps with a different degree, it happens with the write operations, but write operations also reduce the likeliness for a replica of the data in the other DCs as it incurs synchronisation cost. Eventually, these DCs will not have any replica of the data. Given that we do not want to keep replicas if it is not necessary, the need for such a replica will decay as time passes. However, it must be guaranteed that the data is present (replicated) at least in a pre-set minimum number of DCs.

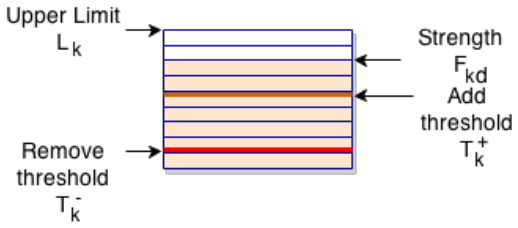



**Figure 1.** Constraints.

For a formal model, we associate each data item $k$ ( $D_k$) in DC $d$ ($DC_d$) with some strength factor $F_{kd}$. A data item is replicated in a DC when its strength exceeds a threshold, $T_k^+$, whereas any record of data $k$ with strength factor below the deletion threshold, $T_k^-$, implies that the data should not be replicated anymore in DC $d$, but only when the other constrains are still kept, i.e. minimum number of replicas needs to be guaranteed. A DC may maintain strength factors of data without keeping a replica of the data item.

Equation 1 describes how read and write accesses influence a local strength factor. Let $X_{dk} = 1$ represent the existence of a replica of data $k$ in DC $d$, or its absence $X_{dk} = 0$. $R_k$ denotes the number of replicas of data $k$ which must be at least $N_k$ (minimum number of replicas).

$$F_{kd} = \max\left(0,\ \min\left(\underbrace{L_k}_{\substack{\text{max.}\\ \text{strength}}},\ \underbrace{r_{kd} * \Delta r_k}_{\text{own reads}} + \underbrace{w_{kd} * \Delta w_k}_{\text{own writes}} - \underbrace{\sum_{i=1, i\neq d}^{|DC|} w_{kd} * \Delta w_{kdi}}_{\text{other writes}} - \underbrace{X_{kd} * \Gamma_k}_{\text{time decay}}\right)\right) \quad (1)$$

Equation 1 shows that the replication strength for the data $k$ in DC $d$ is increased by the reads ($\Delta r_k$) and writes ($\Delta w_k$) requested through DC $d$, with intensities $\Delta r_k$ and $\Delta w_k$, respectively. It is weakened by the writes requested through other DCs than DC $d$, with intensity $\Delta w_{kdi}$. The strength is furthermore weakened by a temporal decay ($\Gamma_k$ , last term in the equation). Equation 1 also states that only DCs with a replica of the data, $D_k$, will be penalised by writes to other DCs and the pass of time ($\Gamma_k$). This reflects the cost of sending updates to the DCs keeping a replica which should be minimised by the strategy. Reads may increase the number of replicas where writes may strengthen the replication in a DC, but may also potentially remove a replica from one of the other DCs, an effect that it is strengthen by the temporal decay of the replication strength.

Each DC with a replica of the data must know about the other DCs having replicas of the same data item. If data item $k$ is only replicated in one DC $d$ and a write is requested using a different DC $j$ than the one it is currently replicated in, so that its replication strength in DC $d$ is falling below $T_k^-$, then the data will continue to be replicated in that DC until a replica is placed in another DC as otherwise the data item will be lost.

The decay factor $\Gamma_k$ ensures that in absence of writes, if the reads are concentrated in a few DCs then the replicas in other DCs with replicas will eventually vanish. This also must comply with the minimum number of replicas, so in some cases, the temporal effect must be ignored to ensure the data exists at least in the minimum number of DCs stipulated.

A read request to a DC, which does not have a replica of the data, will be forwarded to the closest DC with a replica. The DC with a replica will not gain strength from

this read operation as the read was not initiated at this DC. The DC where the read requested originated from will then gain some knowledge about the data access pattern even if it does not have a replica. This knowledge will be used in subsequent reads/writes to the DC and eventually may lead to the placement of a replica. Otherwise, the record will be discarded. Once the replication strength is higher than the replication threshold ($T_k^+$), this DC will notify all the DCs holding a replica about the existence of the new replica on this DC.

### 3.1 Cost of Representation

The execution of an strategy incurs a cost. For the proposed algorithm, the cost is composed of a direct cost associated to the reads ($c_{kd}^r$) and writes ($c_{kd}^w$) executed in the system for the data $k$ in the DC $d$. Also there is a cost associated to the reads ($c_{kdj}^r$, $c_{kdd}^r = 0$) and writes ($c_{kdj}^w$, $c_{kdd}^w = 0$) between DCs $d$ and $j$ for data $k$. $X_{kd}^{'}$ is the new replication state after all the current operations have been executed, with a cost per notification of $c_{*kdj}^r$, and the cost of removing a replica from a DC per notification of $c_{*kdj}^r$. $\rho_{kd}(t_i, t_{i+1})$ is 1 if exists an operation from time $t_i$ to $t_{i+1}$, $(r_{kd}(t_i, t_{i+1}) + w_{kd}(t_i, t_{i+1})) > 0$, or zero otherwise.

$$
\begin{aligned}
&Cost_k(t_i, t_{i+1}) = \\
&\sum_{d=1}^{|DC|} \Bigg( \underbrace{r_{kd}(t_i, t_{i+1}) * c_{kd}^r(t_i) + w_{kd}(t_i, t_{i+1}) * c_{kd}^w(t_i)}_{\text{direct cost}} \\
&\quad \underbrace{+ \sum_{j=1}^{|DC|} w_{kd}(t_i, t_{i+1}) * c_{kdj}^w(t_i) * X_{kj}(t_i)}_{\text{update propagation cost}} \\
&\quad \underbrace{+ (1 - X_{kd}(t_i)) * r_{kd}(t_i, t_{i+1}) * \min_{j \in R_k}(c_{kdj}^r(t_i))}_{\text{redirect reads cost}} \\
&\quad \underbrace{+ (1 - X_{kd}(t_i)) * w_{kd}(t_i, t_{i+1}) * \min_{j \in R_k}(c_{kdj}^w(t_i))}_{\text{redirect writes cost}} \\
&\underbrace{+ \sum_{j=1, j \neq d}^{|DC|} (1 - X_{kd}(t_i)) * X_{kj}(t_i) * \rho_{kd}(t_i, t_{i+1}) * c_{*dj}^r(t_i)}_{\text{search cost}} \\
&\quad \underbrace{+ \sum_{j=1, X_{kj}=1}^{|DC|} X_{kd}^{'}(t_i) * (1 - X_{kd}(t_i)) * c_{*kdj}^r(t_i)}_{\text{new replica cost}} \\
&\underbrace{+ \sum_{j=1, j \neq d, X_{kj}(t_i)=1}^{|DC|} (1 - X_{kd}^{'}(t_i)) * X_{kd}(t_i) * c_{*kdj}^r(t_i)}_{\text{remove replica cost}} \Bigg)
\end{aligned}
\tag{2}
$$

The cost between time $t_i$ and $t_{i+1}$, where $i \in [0, n]$ with $t_0 = 0$ and $t_n = T$, is expressed by Equation 2 and the minimum number of replicas is represented by Inequality 3 for time $t$.

$$
\sum_{d=1}^{|DC|} X_{kd}(t) \geq N_k \tag{3}
$$

The cost in Equation 2 corresponds to the cost of the operations for the reads and writes of data $k$ in DC $d$, 'direct cost', plus the cost of propagating the updates, 'update propagation cost', plus the cost of redirecting the reads to the less costly DC with a replica, 'redirect reads cost', plus the cost of searching for a DC with a replica of the data $k$ to forward the request if the directly access DC does not have a replica of such data, 'search cost', the cost of generating a new replica, 'new replica cost', and finally the cost from removing a replica, 'remove replica cost', from time $t_i$ to time $t_{i+1}$. The Equation 2 has a quadratic term that corresponds to the 'search cost'. The total cost up to a time $T$ is expressed in Equation 4.

$$
Cost_k(T) = \sum_{i=0}^{n-1} Cost_k\left(t_i, t_{i+1}\right) \tag{4}
$$

The algorithm proposed does not consider that, when the DC does not contain the requested data and a search is started, only the first reply from the search is processed and DCs with no replica of the data do not reply to the search request. Also many received reads could be combined into one, which read/write would also be finally forward to the closest DC with a replica, which would reduce the cost.

### 3.2 Discussion

Adaptive replication fits naturally into settings where eventual consistency is used. Information about new replicas can then be eventually propagated between the DCs, such that updates eventually reach also this DC and be applied to the new replica. The deletion of replicas requires strong consistency, but as it is not on any critical path, it will not incur latency increase.

In addition to the decay factor $\Gamma_k$, another temporal effect, the Time To Live (TTL), can be employed to enforce that data items will be fully removed after a while. This value may not be applied to data stored in recovery centres and data warehouses which may have their own TTL. Also it would be desirable for data that expires to be copied into those data centres before it is removed from all the DCs. The replication strategy can be further adapted to include other constraints and objectives, some of which we are already considering.

The presented algorithm is optimal in the sense that, when the access pattern stabilises, the total number of replicas required for the reads and writes is minimal with respect to the defined thresholds.

## Acknowledgments

This work has been supported by the project SyncFree (co-financed by the European Commission through the grant agreement number 609551).

## References

[1] C. L. Abad, Y. Lu, and R. H. Campbell. Dare: Adaptive data replication for efficient cluster scheduling. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 159–168, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4516-5. . URL https://wiki.engr.illinois.edu/download/attachments/194990492/cluster11.pdf.

[2] S. Abdul-Wahid, R. Andonie, J. Lemley, J. Schwing, and J. Widger. Adaptive distributed database replication through colonies of pogo ants. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007. . URL http://www.cwu.edu/~andonie/MyPapers/IPDPS%20Long%20Beach%202007%20final.pdf.

[3] P. M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems*, 13:263–304, 1988.

[4] I. Briquemont. Optimising client-side geo-replication with partially replicated data structures. Master's thesis, Louvain-la-Neuve, September 2014. URL http://www.info.ucl.ac.be/~pvr/MemoireIwanBriquemont.pdf.

[5] A. Chanthadavong. Internet of things to drive explosion of useful data: Emc. Technical report, ZDNet, April 2014. URL http://www.zdnet.com/internet-of-things-to-drive-explosion-of-useful-data-emc-7000028376.

[6] Cisco. The zettabyte era-trends and analysis. Technical report, Cisco, June 2014. URL http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.pdf.

[7] X. Dong, J. Li, Z. Wu, D. Zhang, and J. Xu. On dynamic replication strategies in data service grids. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 155–161, May 2008. .

[8] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[9] S. Goel and R. Buyya. Data replication strategies in wide area distributed systems. In R. G. Qiu, editor, *Enterprise Service Computing: From Concept to Deployment*, pages 211–241. Idea Group Inc, 2006. URL http://www.cloudbus.org/papers/DataReplicationInDSChapter2006.pdf.

[10] R. Kingsy Grace and R. Manimegalai. Dynamic replica placement and selection strategies in data grids- a comprehensive survey. *J. Parallel Distrib. Comput.*, 74(2):2099–2108, Feb. 2013. ISSN 0743-7315. . URL http://dx.doi.org/10.1016/j.jpdc.2013.10.009.

[11] A. Liu, Q. Li, and L. Huang. Quality driven web services replication using directed acyclic graph coding. In A. Bouguettaya, M. Hauswirth, and L. Liu, editors, *Web Information System Engineering – WISE 2011*, volume 6997 of *Lecture Notes in Computer Science*, pages 322–329. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24433-9. . URL http://dx.doi.org/10.1007/978-3-642-24434-6_28.

[12] T. Loukopoulos and I. Ahmad. Static and adaptive distributed data replication using genetic algorithms. *J. Parallel Distrib. Comput.*, 64(11):1270–1285, Nov. 2004. ISSN 0743-7315. . URL http://pdf.aminer.org/000/297/337/static_and_adaptive_data_replication_algorithms_for_fast_information_access.pdf.

[13] N. Mohd. Zin, A. Noraziah, A. Che Fauzi, and T. Herawan. Replication techniques in data grid environments. In J.-S. Pan, S.-M. Chen, and N. Nguyen, editors, *Intelligent Information and Database Systems*, volume 7197 of *Lecture Notes in Computer Science*, pages 549–559. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-28489-2. . URL http://dx.doi.org/10.1007/978-3-642-28490-8_57.

[14] S. Naseera and K. M. Murthy. Agent based replica placement in a data grid environement. *Computational Intelligence, Communication Systems and Networks, International Conference on*, 0:426–430, 2009. .

[15] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 290–297, Dec 2007. . URL http://www.researchgate.net/publication/200023090_Boosting_Database_Replication_Scalability_through_Partial_Replication_and_1-Copy-Snapshot-Isolation/links/0deec520a3cdf6504e000000.

[16] K. Tolle, D. Tansley, and A. Hey. The fourth paradigm: Data-intensive scientific discovery [point of view]. *Proceedings of the IEEE*, 99(8):1334–1337, Aug 2011. ISSN 0018-9219. . URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5958175.

[17] S. Venugopal, R. Buyya, and K. Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Comput. Surv.*, 38(1), June 2006. ISSN 0360-0300. . URL http://www.cloudbus.org/reports/DataGridTaxonomy.pdf.

[18] Z. Wang, T. Li, N. Xiong, and Y. Pan. A novel dynamic network data replication scheme based on historical access record and proactive deletion. *J. Supercomput.*, 62(1):227–250, Oct. 2012. ISSN 0920-8542. . URL http://dx.doi.org/10.1007/s11227-011-0708-z.

[19] O. Wolfson. A distributed algorithm for adaptive replication of data. Technical report, Department of Computer Science, Columbia University, 1990. URL http://hdl.handle.net/10022/AC:P:21285.

[20] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1):181–205, Mar. 1991. ISSN 0362-5915. . URL http://academiccommons.columbia.edu/catalog/ac%3A142996.

[21] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, June 1997. ISSN 0362-5915. . URL http://www.cs.uic.edu/~wolfson/mobile_ps/tods-adaptive-replication.pdf.

# D Conflict-free Partially Replicated Data Types

# Conflict-free Partially Replicated Data Types

Iwan Briquemont, Manuel Bravo, Zhongmiao Li and Peter Van Roy
Université catholic de Louvain, Belgium

***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. Nevertheless, 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. 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 the client. 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.**

## 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. It has been proved that clients are sensitive to even small increases of latency [4], [5].

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 rollback situations.

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 clients caches. In consequence, SwiftCloud reduces latency, and enables off-line mode during disconnection periods.

The current specifications of CRDTs do not allow them to be partitioned. Thus, a CRDT replica is assumed to contain the full data structure. We believe partitioned CRDTs can be effectively used to address two relevant issues of current systems:

- CRDTs can easily become heavy data structucture, such as a CRDT that contains the posts of a user wall in a Facebook-like application. In many cases, the user is simply interested in the most relevant posts, according to some criterium. For instance, one may only 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.
- The full replication of CRDTs in clients arises security concerns. By partitioning the CRDTs, applications could precisely decide which data each client stores. This could keep malicious clients from storing sensitive data.

In this paper, we propose a new set of CRDTs that allow 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.

The major contributions of this paper are the following:

- The definition of the new CPRDTs. This includes revisiting the specifications of previously defined CRDTs.
- Extension of SwiftCloud to integrate CPRDTs.
- Extensive evaluation of the performance improvements of CPRDTs in SwiftCloud. This includes the implementation of a Reddit-like [14], [15] application, called SwiftLinks, on top of SwiftCloud.

The remainder of the paper is organized as follows: Section II introduces previous work that we consider relevant to understand this paper; Section III presents a formal definition of

the partitioned CRDTs and the specifications of some of them; Section IV discusses how CPRDTs could be practically used; Section V presents an extensive evaluation of the SwiftCloud extension that includes CPRDTs; Section VI briefly describes preceding related work; finally, Section VII discusses future work and concludes the paper.

## II. Background

### A. CRDTs

Conflict-free Replicated Data Types (CRDTs) are a set of data structures that allow replicas to be updated concurrently and guarantee all replicas will eventually converge to the same state [8], [9]. Traditional approaches include user interference [7], which is cumbersome for users, or basic last-write-wins semantics [16]. Based on different propagation models, there are two types of CRDTs, namely operation-based and state-based.

For operation-based CRDTs, a replica propagates its applied operations to other replicas. Concurrent operations are designed to be commutative, so replicas can deliver concurrent updates in different order and will converge to the same state, without the risk of having conflict. Causal delivery is usually required.

On the other hand, for state-based CRDTs, a replica ships its whole internal state to the rest. Upon arrival, replicas merge both the local and the received states. The merge operation of state-based CRDTs is idempotent, commutative and associative. Therefore, state-based CRDTs have less requirement for the delivery channel compared to op-based CRDTs: messages can be lost, duplicated or out-of-order, but replicas will converge to the same state as long as they have seen the latest states from each other.

### B. SwiftCloud

SwiftCloud is a geo-replicated cloud storage system that stores CRDTs and caches data at clients [10]. It consists of several datacenters that fully replicate data. Clients communicate with the closest datacenter and cache accessed data in its local cache.

SwiftCloud provides transactional causal+ consistency. Transactions are firstly executed and committed in 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.

## III. Conflict-free Partially Replicated Data Types

In this section we present the Conflict-free Partially Replicated Data Types. These new data types are CRDTs that can be partitioned. We believe that partitioning permits a more efficient usage of resources such as memory and bandwidth. This may be critical when thin clients, such as mobile devices or embedded computers, cache the data structures. On the other hand, we believe CPRDTs have other applications. For instance, CPRDTs can be used to enforce fine-grained security policies. Furthermore, they can also be used to provide a way to support data with multiple fidelity requirements to accommodate resource-limited devices while keeping consistency between the fidelity levels [17]. This can be achieved by not replicating less important information on mobile devices.

This poses new challenges: all operations are not enabled on partial replicas, which means new preconditions must be added to ensure correct usage. However, these conditions must not interfere with the convergence of the replicas. Care must be taken as a partial replica may change over time. A partial replica could change the parts it keeps, by choosing to replicate more parts. This has to be carefully done without losing data and still achieving convergence between replicas.

### 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:

- Users maintain a full replica of their wall.
- A user $X$ that reads or posts in user's $Y$ wall replicates user's $Y$ 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 relationship. For instance, Bob can read all the posts of Alice's wall because they are friends. Nevertheless, Charlie can only read public and Bob's posts (friends of friends). Finally, any other user can only read public posts.

CPRDTs have two applications in this scenario: (i) limiting the size of the wall to be replicated, which can lead to a better usage of memory and bandwidth; and (ii) enforcing security policies.

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, 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 should anticipate how users will use the application. While possible in some cases, it makes the application more cumbersome to write. Furthermore, it would be difficult to achieve optimal results since each client may behave differently.

On the other hand, CPRDTs abstract the partitioning from the application. Thus, from the point of view of the programmer, there will only be one logical data structure per wall. We strongly believe this eases developers 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. For instance, Bob might want to look at the posts that Alice and himself made during the last week. On the other hand,

Charlie may want to see all the posts of the last two years. These two requests will end up with completely different parts of the same set. Only with CPRDTs, optimal results can be achieved.

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. For instance, an anonymous user should only replicate public posts. This would 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. The objects can be read using *query* operations and modified using *update* operations. The 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, can not 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 particles, we need to introduce three concepts to understand CPRDTs: *shard set*, *required*, and *affected*.

**shard set** Each replica of a CPRDT $x_i$ 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 $x_i$. The replica only knows the state of the particles in $\mathrm{shard}(x_i)$; therefore, it can only enable query operations that require those particles. Furthermore, the CPRDT replica only needs to receive update operations that affect the particles in $\mathrm{shard}(x_i)$ in order to converge.

There are two special cases: a *full* replica and a *hollow* replica. When $\mathrm{shard}(x_i) = \pi$ then we say that $x_i$ is a *full* replica, and it 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 [18]). A *hollow* replica does not maintain any state. Nevertheless, it can still handle updates, as explained in section III-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 is enabled only if $\mathrm{required}(op) \subseteq \mathrm{shard}(x_i)$. E.g. for the lookup operation of a set, $\mathrm{required}(lookup(e)) = \{e\}$ where $e$ is an element of the set. In case $e \notin shard(x_i)$, the replica will not be able to 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 $\mathrm{affected}(op)$ tells 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-based 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 between replicas)**.** *$x_i$ and $x_j$ have equivalent abstract states if all* `query` *operations q, for which* $\mathrm{required}(q) \subseteq (\mathrm{shard}(x_i) \cap \mathrm{shard}(x_j))$*, 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 state for particles* $\in \mathrm{shard}(x_i) \cap \mathrm{shard}(x_j)$.

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 the applied update operations. While $x_i$ 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 $C_k(x_i|x_j) = \{f \in$

$C_k(x_i)|\,\text{affected}(f) \in (shard(x_i) \cap shard(x_j))\}$. Intuitively, it includes updates from the causal history of $x_i$ that affects 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:* State-based replication is an interesting propagation mechanism because it poses almost no communication requirements, as explained in II. 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 $x_i$, $x_j$,* $C_{k+1}(x_i) = C_k(x_i) \cup \{f \in C_{k'}(x_j)|\,\text{affected}(f) \in \text{shard}(x_i)\}$

The *merge* method used by a replica must only merge the intersection state of its particles with the remote replica and ignore the others.

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 $\text{affected}(f) \notin \text{shard}(x_j)$.

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 the replica.

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 complete the first phase of the update operation without necessarily complete the second phase. For instance, a replica $x_i$, whose $\text{shard}(x_i)$ are particles $a$ and $b$, receives an update operation that affects particle $c$. In this situation $x_i$ can complete the prepare phase, broadcast the downstream operation to the interested replicas, and discard it locally. We named this scenario *blind updates*. It is important to highlight that this cannot happen in state-based replication. Hollow replicas, which have an empty shard, can only do blind updates.

**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* $C_{k+1}(x_i) = C_k(x_i)$.

### D. Specification

In this section, we extend the CRDT specification models.

*1) Creation of a new partial replica:* The creation of new replicas in CRDTs is rather straightforward. The CRDT can simply be copied in its entirety. Nevertheless, in the context of CPRDTs, we want to choose which particles to copy.

In order to solve this problem, we propose a new operation, called $\text{fraction}$, that 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, $\text{fraction}$ can be defined as follows:
$x_j = \text{fraction}(x_i, Z)$, where $Z$ is the set of particles we want to take. The operations ensures that $\text{shard}(x_j) = \text{shard}(x_i) \cap Z$.

Notice that using a set of particles is the canonical form to define the subset. In practice, it can be defined by using a more 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", which can then be transformed into a set of particles. We further discuss this in Section IV-A.

This operation is also useful to simplify the specifications of state-based CRDTs: when merging two partial states, we only want to merge the state of the common particles since the rest can be ignored. However, putting this in the specification is cumbersome. Instead, we assume that the merge operation merges the complete payloads, regardless of their shard. We can then limit the growth of the replica to its own shard as such: if replica $x_j$ receives the payload of replica $x_i$, $x_j$ should do:
$x_k = \text{fraction}(\text{merge}(x_i, x_j), \text{shard}(x_j))$
Thus $\text{shard}(x_k) = \text{shard}(x_j)$ and, in consequence, the replica does not grow. In practice, the fraction operation can be applied before sending the state in order to save bandwidth.

*2) Specification model:* The specifications are similar to the CRDT specifications, with some added notations. Each operation must define which particles it involves (*required* particles and *affected* particles). Note that the conditions given in section III-B ($\text{required}(op) \subseteq \text{shard}(x_i)$, and $\text{affected}(op) \in$

$\text{shard}(op(x_i))$ for state-based replication), regarding whether an operation is enabled or not, are not explicitly included in the specification. Nevertheless, it must be enforced.

Specification 1 and Specification 2 show the template of specifications for state-based and operation-based CPRDTs respectively. Examples of CPRDTs can be found in the appendix.

**Specification 1** State-based CPRDTs specification

1: **particle definition** Informal definition of what is a particle
2: **payload** type
3: **initial** *Initial value*
4: **query** *query(arguments)* : *returns*
5: **required particles** *Set of required particles*
6: **pre** *Precondition*
7: **let** *Evaluate synchronously, no side effects*
8: **update** *update(arguments)* : *returns*
9: **required particles** *Set of required particles*
10: **affected particle** *The particle that can be affected*
11: **pre** *Precondition*
12: **let** *Evaluate at source, synchronously*
13: **merge** ($value1$, $value2$) : payload $mergedValues$
14: *Least Upper Bound merge of* $value1$ *and* $value2$
15: $\text{shard}(mergedValues) = \text{shard}(value1) \cup \text{shard}(value2)$ *must be true*
16: **fraction** (particles $selection$) : payload $partialReplica$
17: *Copies the particles* $selection$ *into* $partialReplica$ *so that* $\text{shard}(partialReplica) = selection \cap \text{shard}(self)$

**Specification 2** Op-based CPRDTs specification

1: **particle definition** Informal definition of what is a particle
2: **query** *query(arguments)* : *returns*
3: **required particles** *Set of required particles*
4: **pre** *Precondition*
5: **let** *Evaluate synchronously, no side effects*
6: **update** *Global update(arguments)* : returns
7: **prepare** (*arguments*) : *intermediate value(s) to pass downstream*
8: **required particles** *Set of required particles to prepare*
9: **pre** *Precondition*
10: **let** *1st phase: synchronous, at source, no side effects*
11: **effect** (*arguments passed downstream*)
12: **required particles** *Set of required particles to apply*
13: **affected particles** *The particle that can be affected*
14: **pre** *Precondition against downstream state*
15: **let** *2nd phase: asynchronous, side effects*
16: **fraction** (particles $selection$) : payload $partialReplica$
17: *Copies the particles* $selection$ *into* $partialReplica$ *so that* $\text{shard}(partialReplica) = selection \cap \text{shard}(self)$

## IV. Practical usage

In this section, we describe some practical issues of CPRDTs . This takes us to discuss three things: (i) how shard can be practically defined, (ii) how replicas of CPRDTs are created and modified through *shard queries*, and (iii) how the shard can be managed in a real system. The last part of the section discusses a system model that aim to simplify and ease the integration of CPRDTs.

### A. Shard definition

In Section III, we defined the shard of a replica as a set of particles. This set can be infinite; therefore, all elements of the set are not explicitly kept in practice as we only need to know whether a particle is in the shard or not. A shard can be then defined as a range of particles. For instance, on a set of integers, we can define it as $[0, 2]$ for particles $\{0, 1, 2\}$, or even $(0, +\infty)$ for strictly positive integers. Similarly, it can be defined as all the particles that satisfy a specific property. For example, the odd integers.

### B. Shard query

A *shard query* defines the set of particles that satisfy a particular criterium. Thus, in practice, *shard queries* can be used for two reasons: (i) creation of new CPRDTs from the returned set of particles, and (ii) shrinking or expanding the *shard set*. The latter is discussed in more detail in IV-C.

*Shard queries* bridge the gap between the application semantics and the function fraction, introduced in III-D1. Thus, it adds expressiveness to the usage of CPRDTs.

We identify two types of *shard queries*: state-independent and state-dependent queries. The former only depends on the properties of the particles, and not in the state of CPRDT. In contrast, the latter depends on the current version of the CPRDT. For instance, a state-independent query over a set of integers could be "integers greater than 0". On the other hand, a state-dependent query could be "10 highest integers in the set". The state-independent query does not depend on the state of the CPRDT, and the result of the query will always be the set $(0, +\infty)$. Nevertheless, the state-dependent query will have a different result depending on which elements have been already added, and removed, on the version considered.

State-independent queries are easier to work with: they are comparable. One could determine which query is more specific without having to know the state of the CPDRT they apply to. While with state-dependent queries, one can only compare queries if they apply to the same version of the object. Nevertheless, we believe both types are needed in order to make CPRDTs usable. In IV-D, we describe a system model that can simplify the integration of state-dependent of queries.

### C. 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 very useful in practice. For instance, a client may become interested in new parts. Having dynamic *shard set*, the 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 one hand, a partial replica can easily shrink its *shard set* without compromising convergence in the operation-based scenario. The replica only needs to take into consideration two things: (i) updates prepared locally have been already broadcasted, and (ii) the data to be dropped is replicated by some other replica; therefore, data do not disappear. On the other hand, expanding a partial replica is

more tricky. For instance, in an operation-based scenario, the following situation can easily occur:

- A replica's ($x_i$) *shard set* is $a, c$.
- $x_i$ did not receive updates that affect $b$ for a while.
- Suddenly, $x_i$ becomes interested in $b$ and starts accepting updates on $b$.
- Unfortunately, the replica will not converge since updates have been missed.

In the previous scenario, extra communication between replicas would be needed in order to recover dropped updates. This is clearly not easy to achieve.

In state-based replication, shrinking or expanding the *shard set* is simpler. In one hand, a replica only needs to broadcast its state before expanding 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 the new particles.

### D. Authority system model

We have not specified a system model up to now. We have only said that processes storing objects propagate either states or operations to reach convergence. CPRDTs are not biased to any specific system model. Nevertheless, we believe that certain system models might considerably simplify the managment of partial replicas.

We propose a system model where there is at least one entity, namely authority, holding a full replica. This model poses several advantages in comparison to an ad-hoc architecture where no authority is assumed. Firstly, clients can discard their (partial) replicas at will as long as their updates have been reliably sent to the authority. Secondly, a client can request any fraction to the 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.

## V. 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. Efficient resources usage positively impacts the performance. In our study we compare three approaches: (i) classical geo-replicated system where data is exclusively stored in datacenters, (ii) SwiftCloud, and (iii) our modified version of SwiftCloud that integrates CPRDTs.

**SwiftLinks** In order to compare the three systems, we implemented a new application, namely SwiftLinks, on top of SwiftCloud. SwiftLinks is a vote-based content-sharing application based on Reddit. In few words, the application allows users to create forums where they can publish posts. Then, users can vote positively or negatively posts. As a consequence, posts get ranked according to the votes. In addition, users can add comments to posts and to other comments. Users can also vote comments, and consequently the comments get ranked (more information [14], [15]).

SwiftLinks is modeled with three CRDTs: (i) OR-Set for forums, (ii) a novel Remove-once Tree CRDT for comments, and (iii) Last-Writer-Wins Registers for votes. The application uses both types of queries: state-independent and state-dependent. The former is mostly used for reading single comments or posts. The latter is used for reading raking 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 operation: (i) new post, (ii) new comment, and (iii) new vote.

For most of the experiments, 20% of the operation are updates and 80% are read operations. 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. 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. Each client performs an operation per second.

There are three main configurations for clients to run the application: *cloud*, *non-lazy*, and *lazy*.

In the *cloud* configuration, operations are applied synchronously at one datacenter and replicated asynchronously to the rest of datacenters. This simulates a typical geo-replicated system. In this case, the client does not cache any data.

The other two configurations adopt the SwiftCloud approach of caching data on clients side. We limit the capacity of the cache in our experiments, using 64MB as default size. If the cache size exceeds this limit, the least recently used object is dropped. This simulates memory restrictions on thin clients. In this configurations, *non-lazy* and *lazy*, if the cache contains the required data, the operations are run locally, and asynchronously propagated to the closest datacenter.

The difference between *non-lazy* and *lazy* 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 *non-lazy* configuration, the objects are only fully replicated

in clients side, as in SwiftCloud. Therefore, the same top ten posts query would replicate the whole forum.

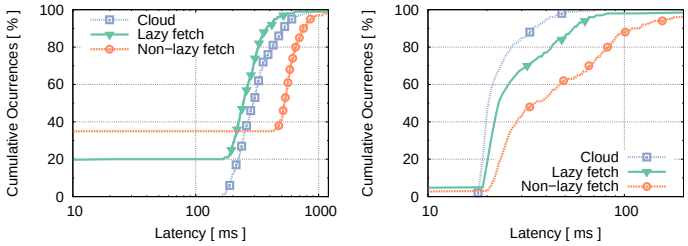
### B. Latency

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 with a 64MB cache size limit. 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 *non-lazy* and *lazy* 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 *non-lazy* mode has greater cache hit rate (35%) than the *lazy* 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 2 shows the results of a similar experiment but without any cache size limit. In that case, the cache hit rate, for the *non-lazy* 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 *lazy* mode, the cache hit rate is lower, with only 20% in both experiments (figures 1a and 2), 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 lazy fetching, 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.

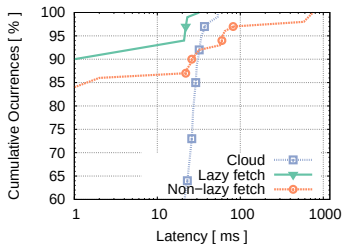
For the latency of reading comments of a post, shown in Figure 1b, the situation is a bit different. Clients are less likely to read the same comment tree multiple times; therefore, this affects the cache hit ratio. As the figure shows, the hit ratio is less than 5% in both *lazy* and *non-lazy* fetching. But again, *lazy* fetching 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 *lazy* approach has a slightly better latency, close to the *cloud* mode. The *cloud* mode performs better because it does never need to fetch any data, which means the returned messages are considerable smaller. Notice that the difference between *non-lazy* and *lazy* mode has been reduced in this experiment because the involved objects are smaller.

**Update operations** Caching modes (*lazy* and *non-lazy*) 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 state-independent queries to




(a) Reads of pages of links

(b) Reads of comments of a link




(c) Updates: posting a link, commenting, voting a link, and voting a comment

Fig. 1: Perceived latency of SwiftLinks at one site with medium (64MB) cache size limit and a warmed up cache.

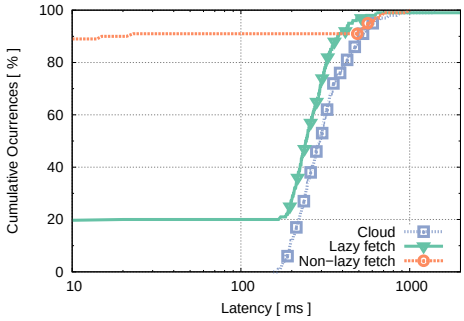



Fig. 2: Reads of pages of links with unlimited cache which is already warmed up.

fetch their missing part, which substantially simplifies the comparison of partial objects in the cache. Figure 1c 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 *lazy* mode has the advantage of reducing the latency when the cache is not hit, as it only needs to fetch the part of the object that needs to be updated, instead of the full object. Moreover, some updates can be done blindly, therefore, they are completed locally.

In particular, Figure 3 shows the benefit of updates when posting comments, which almost always only requires particles already present in the cache. One can see that with lazy fetching, all the operations have almost no latency, as they can be done completely asynchronously. In contrast, in *non-lazy* mode, there can be a large delay when the tree of comments

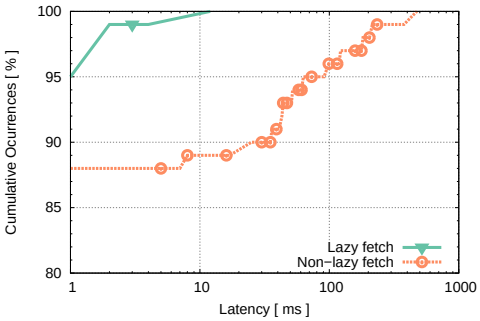



Fig. 3: Perceived latency of commenting on a post, at all sites, with medium (64MB) cache size limit.

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 *lazy* mode, the client only has to fetch part of the tree to complete the update.

### C. Impact of cache size limit

In this section we look at how the application performance changes with various cache size limits (16MB, 64MB, and 128MB).

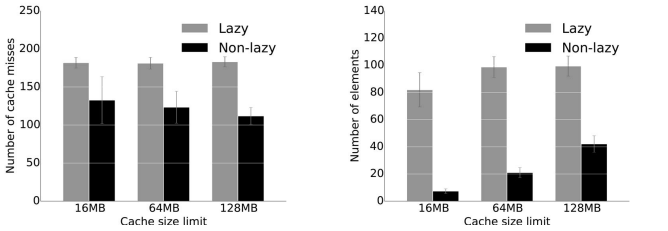
*1) Impact on latency:* We have already proved that the *non-lazy* mode performs better without cache limit when reading links. We run the same experiments showed in Figure 1 setting the cache size limit to 16MB and 128MB. We do not show the plot due to space restrictions.

The experiments show that a smaller cache (16MB) size limit has a big latency impact on reading links and updates in *non-lazy* mode. Nevertheless, its impact is considerable smaller in *lazy* mode. With a small cache, the cache hit rate of *non-lazy* mode of reading links becomes worse than in *lazy* mode. This is caused because only 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 *lazy* mode, but it is considerable lower: it only drops to 13% from 20%. The same applies for update operations.

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 *non-lazy* mode has a large portion of zero latency operations when reading links, as more link sets can be kept in the cache. It however still performs worse than *lazy* fetch for operations that do not hit the cache. The latency of update operations is also improved for the *non-lazy* mode with a bigger cache, but the *lazy* mode still outperforms it for the same reasons.

*2) Impact on cache miss rate:* The size limit imposed on the cache has an impact on the cache hit rate. Figure 4a shows that the *lazy* mode is less impacted by the cache size limit than the *non-lazy* mode. With the three cache limits,




(a) Number of cache misses with different cache size limits.

(b) Total number of objects kept in the cache.

Fig. 4: Impact of cache size limit in lazy and non-lazy modes

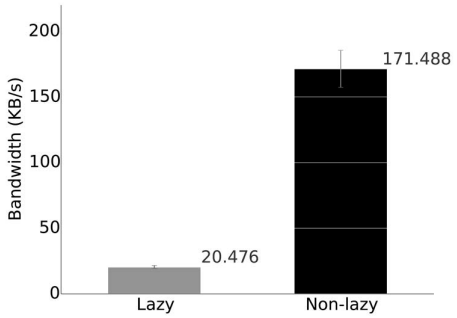



Fig. 5: Average bandwidth usage to fetch objects with a 128MB cache limit, with the cache already warmed up.

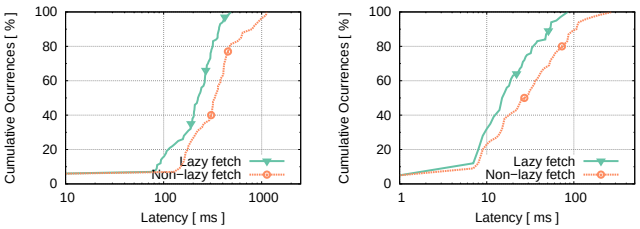
the *lazy* mode shows a rather stable number of cache misses, about 180. Nevertheless, this does not apply to the *non-lazy* mode, where the number of caches misses increases as the cache size limit is reduced. As in previous experiments, the number of cache misses is always greater in the *lazy* mode. Nevertheless, we have already proved that the latency in *lazy* mode, is always smaller in average.

*3) Impact on number of objects in the cache:* Another impact of the cache size limit is 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 4b shows the difference between both modes: *lazy* and *non-lazy*. In the *lazy* mode, many more objects can fit in the cache at any moment, since only parts of the object are kept. 64MB is enough to keep all the objects needed by the application, while in the *non-lazy* mode, even 128MB is not enough. This leads us to determine that the *lazy* mode makes a more intelligent use of the cache, allowing more object to coexist at the same time.

### D. Bandwidth usage

In order to measure the bandwidth usage, we measure the average bandwidth usage of one client over one minute, with the cache already warmed up. Figure 5 compares the *lazy* and the *non-lazy* modes. As the figure shows, the *lazy* reduces significantly the bandwidth used by a client.




(a) Reads of pages of links (b) Reads of comments of a link

Fig. 6: Perceived latency of SwiftLinks at one site during cache warm up.

### E. Cache warm up

The results shown previously are taken with the cache warmed. In practice, this will not always be the case. The following experiments compare both *lazy* and *non-lazy* modes latencies when the cache is still cold, i.e. no objects are stored in the client side.

Figure 6 shows the latency of operations during the first 10 seconds of running the application, with a cold cache. In this case, the *lazy* mode produces lower latencies as it does not need to replicate the full object. The difference is more noticeable for links reading operations, as shown in Figure 6a, as the set of links are large objects. But even for smaller objects, such as comment trees, the lazy mode outperforms the *non-lazy* one (Figure 6b). It is important to notice that the cache size limit is not impacting these experiments, since after 10 seconds, the cache does not get full.

### F. Discussion

We have seen that lazy fetching has advantages over full replication of objects. 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 in the cache.

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.

Partial replication also allows to reduce the bandwidth usage of the application by a factor of 8, which can be especially valuable on mobile wireless connections, such as EDGE or 3G.

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. Lazy fetching limits this issue by only replicating the parts of the object that are actually needed.

Unfortunately, lazy fetching has one main drawback, it limits the cache hit rate, as an object is not fully replicated right away, and non-replicated parts may be needed by subsequent operations. Therefore, the *lazy* mode should be used depending on the workload and the cost of a cache miss. Nevertheless, a trade-off is possible between the two: instead of only fetching the parts needed by the operations, we could fetch more parts of the object in order to improve the cache hit rate. This would however increase bandwidth and cache size utilisation. Latency could be kept low by doing this additional fetch asynchronously, when the user is not doing any operation.

## VI. Related work

**Optimizing memory and bandwidth usage for CRDTs** Bandwidth and space usage of CRDTs is a concern in the research community. Burckhardt et al. [19] formally calculate the space requirements for different replicated data types, such a state-based counter and a state-based set.

On the other hand, Bieniusa et al. proposed an optimization for CRDT sets that can avoid the use of tombstone by using vector clock to capture causal history [20]. Thus, the state kept by the CRDT is considerably reduce.

Finally, Almeida et al. proposed Delta-state conflict-free replicated data types [21] that allows state-based CRDT to only propagate partial states instead of the whole state. While this approach improves bandwidth usage, it does not reduce the storage space for CRDTs.

**Partial replication** PRACTI [22] 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 [23] 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 [24], 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.

## VII. 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. We have explained how state-based and operation-based replication mechanisms should be adapted to support partial replicas. We have also shown how to specify CPRDTs by building upon previous work. Moreover, we have given examples of CPRDTs such as a state-based grow-only set and a grow-only tree.

In order to evaluate our solution, we have integrated CPRDT into SwiftCloud, a geo-replicated storage system that replicates CRDTs on client-side in order to reduce latency. We have also

implemented a Reddit-like application, called SwiftLinks, on top of the modified version of SwiftCloud. In our evaluation, we have compared three scenarios: classical geo-replicated storage system, SwiftCloud and SwiftCloud with CPRDTs.

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. The experimental study has also shown that CPRDTs reduce the latency in average in comparison to the full replication scenario. However, CPRDTs have a negative impact on the cache hit rate, which has to be weighted against the upper bound on the latency that provides.

We plan to extend this work in several directions. Firstly, we want to evaluate CPRDTs in different scenarios. This would imply implementing different kind of applications on top. This would help us to get an even better view of its benefits and drawbacks. Secondly, as we already mentioned in the introduction of the paper, partial replication can be used as a security mechanism to avoid replicating sensitive data by restricting access with finely grained rules. We believe is an interesting way of exploiting CPRDTs. Finally, we want to study how predictive caching techniques could still improve bandwidth usage and consequently reduce latency even more.

The code of our modified version of SwiftCloud can be found in *XXX*.

## ACKNOWLEDGMENT

We thank Marek Zawirski for his help integrating CPRDTs into SwiftCloud. This work was partially funded by the XXX project in the European Seventh Framework Programme (XXX) under Grant Agreement n$^o$ XXX and by the XXX under Grant Agreement XXX.

## 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. [Online]. Available: http://doi.acm.org/10.1145/2491245

[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 *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 401–416. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043593

[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. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[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. [Online]. Available: http://doi.acm.org/10.1145/1275511.1275514

[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. [Online]. Available: http://dl.acm.org/citation.cfm?id=224070

[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. [Online]. Available: http://hal.inria.fr/inria-00555588

[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, X. Dfago, F. Petit, and V. Villain, Eds. Springer Berlin Heidelberg, 2011, vol. 6976, pp. 386–400. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24550-3_29

[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] reddit inc, "About reddit," http://www.reddit.com/about/, accessed: 2014-06-02.

[15] ——, "reddit source code," https://github.com/reddit/reddit, accessed: 2014-04-08.

[16] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

[17] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *Mobisys 2009: Proceedings of the 7th international conference on Mobile systems, applications, and services*. Association for Computing Machinery, Inc., June 2009.

[18] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro, "Incremental stream processing using computational conflict-free replicated data types," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, ser. CloudDP '13. New York, NY, USA: ACM, 2013, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/2460756.2460762

[19] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: Specification, verification, optimality," in *41st Symposium on Principles of Programming Languages (POPL)*. ACM SIGPLAN, January 2014.

[20] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte, "An optimized conflict-free replicated set," *ArXiv e-prints*, Oct. 2012.

[21] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by decomposition," in *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, ser. PaPEC '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:2. [Online]. Available: http://doi.acm.org/10.1145/2596631.2596634

[22] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "Practi replication," in *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267680.1267685

[23] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '09. New York, NY, USA: ACM, 2009, pp. 83–94. [Online]. Available: http://doi.acm.org/10.1145/1555816.1555826

[24] 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 *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 261–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558995

[25] I. Briquemont, "Optimising client-side geo-replication with partially replicated data structures," Sep. 2014.

## APPENDIX
## CPRDTS EXAMPLES

In this section we propose the specifications for some CPRDTs. We mostly adapt the CRDT specifications proposed by Shapiro et al ( [8], [9]). We also introduce a tree CPRDT. To the best of our knowledge, a tree CRDT has never been formally specified before. More CPRDTs specifications can be found in [25].

**Grow-Only set** Specification 3 gives a simple state-based grow only set, which only supports the add operation).

**Specification 3** State-based Grow-Only Set (G-set) with Partial Replication

**particle definition** A possible element of the set.
**payload** set $A$
  **initial** $\varnothing$
**query** lookup(element $e$) : boolean $b$
  **required particles** $\{e\}$
  **let** $b = e \in A$
**update** add(element $e$)
  **required particles** $\varnothing$
  **affected particles** $\{e\}$
  $A := A \cup \{e\}$
**merge** ($S$, $T$) : payload $U$
  **let** $U.A = S.A \cup T.A$
**fraction** (particles $Z$) : payload $D$
  **let** $D.A = A \cap Z$

**Grow-only tree** A state-based grow-only tree is specified in Specification 4. A node is defined by its path and its content in a recursive way, which is noted as $(parent, nodeContent)$, where $parent$ is defined similarly. The root is represented by empty: $()$. For instance, a node $(((), 1), 2)$ has content 2 and parent $((), 1)$. This allows to make a grow-only tree that is very similar to a set, with only the added precondition that the parent must exist when adding a node. It also means that adding nodes which have the same value and the same parent result in one node in the tree.

The particles for this tree are the nodes, i.e. their parents and their content.

**Specification 4** State-based Grow-Only Tree (G-tree) with Partial Replication.

**particle definition** A node of the tree.
**payload** set $A$
  **initial** $\varnothing$
**query** lookup(node $n$) : boolean $b$
  **required particles** $\{n\}$
  **let** $b = n \in A$
**update** add(node $(parent, content)$)
  **required particles** $\{parent\}$ (if parent is not the root)
  **affected particles** $\{(parent, content)\}$
  **pre** $parent \in A$
  $A := A \cup \{(parent, content)\}$
**merge** ($S$, $T$) : payload $U$
  **let** $U.A = S.A \cup T.A$
**fraction** (particles $Z$) : payload $D$
  **let** $D.A = A \cap Z$

**Observed-Removed set**

In the Specification 5, we show the CPDRT specification of an Observed-Removed set (its original formalization can be found in [8]). It is an operation-based specification that assumes causal delivery of its operations to optimise the metadata size. A particle is defined as an element of the set. Each added element is internally uniquely tagged. When removing an element, only associated unique tags observed at the source replica are removed, so a remove operation does not affect a concurrent add operation.

Notice that the add operation can be a blind update: it does not require any particle in the prepare phase, and it can thus be prepared by a replica which does not have the element to be added in its shard. The remove operation requires the particle of the element it removes, as it needs to send the added $(e, u)$ pairs it observed to the other replicas.

**Specification 5** Op-based Observed-Remove Set (OR-set) with Partial Replication

**particle definition** A possible element of the set.
**payload** set $S$
  **initial** $\varnothing$
**query** lookup(element $e$) : boolean $b$
  **required particles** $\{e\}$
  **let** $b = \exists u : (e, u) \in S$
**update** add(element $e$)
  **prepare** $(e)$ : $\alpha$
    **let** $\alpha = unique()$
  **effect** $(e, \alpha)$
    **affected particles** $\{e\}$
    $S := S \cup \{e, \alpha\}$
**update** remove(element $e$)
  **prepare** $(e)$ : R
    **required particles** $\{e\}$
    **pre** lookup($e$)
    **let** $R = \{(e, u) | \exists u : (e, u) \in S\}$
  **effect** $(R)$
    **affected particles** $\{e\}$
    **pre** $\forall (e, u) \in R : add(e, u)$ has been delivered
    $S := S \setminus R$
**fraction** (particles $Z$) : payload $D$
  **let** $D.S = \{(e, u) \in S | e \in Z\}$

# E A Study of CRDTs that do computations

# A Study of CRDTs that do Computations

David Navalho Sérgio Duarte Nuno Preguiça
NOVA LINCS, FCT, Universidade NOVA de Lisboa

## Abstract

A CRDT is a data type specially designed to allow instances to be replicated and modified without coordination, while providing an automatic mechanism to merge concurrent updates that guarantees eventual consistency. In this paper we present a brief study of computational CRDTs, a class of CRDTs whose state is the result of a computation over the executed updates. We propose three generic designs that reduce the amount of information that each replica maintains and propagates for synchronizations. For each of the designs, we discuss the properties that the function being computed needs to satisfy.

## 1. Introduction

Cloud infrastructures, composed of multiple data centers spread across the globe, have become central for the deployment of novel Internet services, from social networks to business applications. A large number of cloud databases have been developed in recent years, providing different level of consistency, from strong [5] to eventual consistency [2, 6, 7].

In this paper we focus on cloud databases that provide eventual consistency only. When using an eventually consistent database, applications can be made highly available by replicating the application code and data in multiple data centers and allowing a user to access any of these data centers. Low latency is achieved by routing the client requests to the closest data center and executing the request in the data center without coordinating with other data centers.

In such settings, concurrent updates may be executed in different replicas. Systems must provide a mechanism to handle concurrent updates and enforce eventual convergence of all replicas. CRDTs [10] have been proposed as a technique for helping application programmers to deal with concurrent updates. They provide eventual consistency with well defined semantics and thus make these systems more amenable to programmers. CRDTs have been adopted as a key feature in a leading cloud database, Riak, and are used in multiple large-scale systems, such as SoundCloud and Twitter's Summingbird[3].

Most CRDTs proposed in literature are replicated forms of collections. In such data types, a replica needs to maintain all data elements in all replicas. Thus, a model where every data replica maintains the same state and where all updates are propagated to all replicas is natural.

In some cases, applications are not interested in actual elements or updates, but instead on the result of a computation over them. We call computational CRDTs to the class of CRDTs whose state is the result of a computation over the executed updates. For example, a counter CRDT [10] counts the number of times an increment operation has been executed. In such cases, each replica does not need to maintain every individual update, but can instead maintain for each replica an integer that counts the number of increments executed at that replica. For synchronizing replicas, it also suffices to propagate an integer that summarizes a set of updates.

In the remaining of this paper we present a brief study of the properties of computational CRDTs. In particular, we propose three generic designs that minimize the data that needs to be maintained in each replica and that needs to be propagated for synchronizing replicas. We study the properties of functions suitable to each of the designs. Notably, our last design departs from the strict model of state-based CRDTs by the fact that the state of each replica does not need to converge, although the result of all queries executed in every replica is the same.

### 1.1 Related work

Aggregation techniques have been studied extensively in different settings, such as as sensor networks [11]. Our work can build on the proposed algorithms for creating replicated data types that perform computations in a cloud database.

The techniques used to model CRDTs have been used to express a distributed deterministic dataflow model for concurrent communication between processes [8]. They have also been used to provide algebraic structures for integration between batch and stream processing of aggregations [3] and to support incremental computations [9]. Unlike these works, this paper studies CRDTs that can be integrated in a cloud database as an elementary abstraction to perform computations without requiring additional support from the system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*PaPoC'15*, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3537-9/15/04. . . $15.00.
http://dx.doi.org/10.1145/2745947.2745948

The problem of optimizing information propagated for synchronizing replicas has been studied by Almeida et. al [1], who have proposed a principled approach to merge the changes produced by multiple operations and use this information to update a remote replica. In our work, the information propagated to synchronize replicas also summarizes multiple updates. However, all information is handled in the context of the CRDT. Additionally, our last design departs from the strict state-based CRDT model by allowing replicas to maintain different state.

## 2. System model

We adopt the CRDT state-based model [10], where replicas synchronize in a peer-to-peer way, by sending their state to other replicas, where the received state is merged with the current state. A CRDT has an interface that includes update operations that modify the state of the object. In our presentation, we define an event as an invocation of an update operation. For simplicity, we consider a single read-only operation that returns the state of the object. A CRDT includes an additional operation, *merge*, to merge a copy of a remote replica with the current replica state. In one design, we extend this model to allow a replica to send only a subset of its state to other replicas.

For fault-tolerance, we assume a crash-recovery model, where a replica that fails recovers with its state intact. In a typical cloud deployment, each data center can be seen as a single replica, although internally an object is replicated in a quorum of replicas.

## 3. Design 1: Incremental Computations

Our first design considers computations that can be done incrementally. In this case, computing the function over two disjoint sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental if there is a function fun, such that:

$$\mathcal{F}^{\mathtt{fun}}(E_1 \cup E_2, \mathsf{hb}_{E_1 \cup E_2}) = \mathtt{fun}(\mathcal{F}^{\mathtt{fun}}(E_1, \mathsf{hb}_{E_1}), \mathcal{F}^{\mathtt{fun}}(E_2, \mathsf{hb}_{E_2}))$$

where $E_1$ and $E_2$ are disjoint sets of events (operation invocations), $\mathsf{hb}_E$ is a partial causality order on $E$[1], and $\mathcal{F}^{\mathtt{fun}}$ is the function that defines the state of a CRDT that computes $fun$ over the observed events (following loosely the formalization proposed by Burckhardt et. al.[4]).

For example, a counter with a single update operation for increment, inc, can be defined as follows:

$$\begin{aligned} \mathcal{F}^{+}_{\mathtt{ctr}}(E, \mathsf{hb}) &= |\{e \in E : e = \mathtt{inc}\}| \\ \mathcal{F}^{+}_{\mathtt{ctr}}(E_1 \cup E_2, \mathsf{hb}) &= \mathcal{F}^{+}_{\mathtt{ctr}}(E_1, \mathsf{hb}) + \mathcal{F}^{+}_{\mathtt{ctr}}(E_2, \mathsf{hb}) \end{aligned}$$

For these computations, Figure 1 presents a generic CRDT design that is parameterized by the following elements: (i) $V_0$, the initial state associated with a replica; (ii) $\mathtt{fun}(o)$, the value of the computation for a single operation $o$; (iii) $\mathtt{fun}(s_1, s_2)$, the function to compose two partial results; and (iv) $\mathtt{fun}^{\mathtt{max}}(v_1, v_2)$, that returns the latest of two values.

In this design, each replica computes its contribution to the final value of the CRDT independently. Each replica maintains a map for the contributions of each replica. When executing an update operation, a replica updates its contribution by using function fun to combine the previous computed contribution and the contribution of the new operation (with $s[i \mapsto \mathtt{fun}\big(s[i], \mathtt{fun}(\mathtt{op})\big)]$ representing the replacement in $s$ of the value of entry $i$ by the new computed value). When merging two replicas, for the partial result of each replica, the most recently computed result must be kept, which is returned by $\mathtt{fun}^{\mathtt{max}}$. If the values are monotonic, it is immediate to know what is the most recent version. Otherwise, it might be necessary to maintain this information explicitly. The value of a replica is computed by applying the function fun to the contributions of all replicas.

As an example, a positive-negative counter, with an increment and a decrement operations can be defined by making:

$$\begin{aligned} V_0 &= (0,0) \\ \mathtt{fun}(\mathtt{inc}) &= (1,0) \\ \mathtt{fun}(\mathtt{dec}) &= (0,1) \\ \mathtt{fun}((p,m),(p',m')) &= (p+p', m+m') \\ \mathtt{fun}^{\mathtt{max}}((p,m),(p',m')) &= (max(p,p'), max(m,m')) \end{aligned}$$

A CRDT that computes the average of values added to an object, which could be used for example to present the average rating in a web application, can be defined by making:

$$\begin{aligned} V_0 &= (0,0) \\ \mathtt{fun}(\mathtt{add}(x)) &= (x,1) \\ \mathtt{fun}((s,c),(s',c')) &= (s+s', c+c') \\ \mathtt{fun}^{\mathtt{max}}((s,c),(s',c')) &= (s,c), \text{iff } c > c' \\ &\quad\ (s',c'), \text{iff } c \leq c' \end{aligned}$$

The average is computed as $s/c$, with $(s, c)$ the result of the read defined in the generic CRDT design.

Other CRDTs can be defined using a similar approach, including a CRDT that computes a histogram.

## 4. Design 2: Incremental Idempotent Computations

In some cases, the computation to be performed besides being incremental is also idempotent. In this case, computing the function over two (potentially overlapping) sets of events and combining the results is equal to computing the function over the union of the two sets. Formally, a computation is incremental and idempotent if there is a function fun, such that for any sets of events $E_1$ and $E_2$ we have:

$$\mathcal{F}^{\mathtt{fun}}(E_1 \cup E_2, \mathsf{hb}) = \mathtt{fun}(\mathcal{F}^{\mathtt{fun}}(E_1, \mathsf{hb}), \mathcal{F}^{\mathtt{fun}}(E_2, \mathsf{hb}))$$

[1] For simplicity of presentation, we drop the subscripts of hb in the rest of the paper.

| | | | |
|---|---|---|---|
| Replica state | $\Sigma$ | $=$ | $\mathbb{I} \to V$ |
| Initial state | $\sigma_i^0$ | $=$ | $V_0$ |
| Update op at replica $i$ | $\mathtt{op}_i(s)$ | $=$ | $s[i \mapsto \mathtt{fun}(s[i], \mathtt{fun}(\mathtt{op}))]$ |
| Read at replica $i$ | $\mathtt{op}_i(s)$ | $=$ | $\mathtt{fun}(s[i], \forall i)$ |
| Merge replica states | deliver $(s, s')$ | $=$ | $s[i \mapsto \mathtt{fun}^{\mathtt{max}}(s[i], s'[i])], \forall i$ |

Figure 1: Generic CRDT for incremental computation.

| | | | |
|---|---|---|---|
| Replica state | $\Sigma$ | $=$ | $V$ |
| Initial state | $\sigma_i^0$ | $=$ | $V_0$ |
| Update op at replica $i$ | $\mathtt{op}_i(s)$ | $=$ | $\mathtt{fun}(s, \mathtt{fun}(\mathtt{op}))$ |
| Read at replica $i$ | $\mathtt{op}_i(s)$ | $=$ | $s$ |
| Merge replica states | deliver $(s, s')$ | $=$ | $fun(s, s')$ |

Figure 2: Generic CRDT for incremental idempotent computation.

For these computations, Figure 2 presents a generic CRDT design. In this case, it is possible to keep in each replica only the computed result that is modified when executing update and merge operations.

A computation that obeys these conditions is computing the maximum of the values added to an object, which could be used in a game application for keeping the highest score. This data type could be implemented, keeping a name associated with the highest score, with names totally ordered, by making:

$$\begin{aligned} V_0 &= (-, \text{minimum value}) \\ \mathtt{fun}(\mathtt{add}(n, v)) &= (n, v) \\ \mathtt{fun}((n, v), (n', v')) &= (n, v), \text{ iff } v > v' \vee (v = v' \wedge n > n') \\ &\quad\ (n', v'), \text{ otherwise} \end{aligned}$$

A generalization of the maximum CRDT is a top-K CRDT that keeps the K players with highest scores, which can be used to maintain a leaderboard in a game application. This CRDT can be implemented by making:

$$\begin{aligned} V_0 &= \{\} \\ \mathtt{fun}(\mathtt{add}(n, v)) &= \{(n, v)\} \\ \mathtt{fun}(s, s') &= \mathtt{max_k}\big(\{(n, v) \in (s \cup s') : \\ &\qquad \nexists (n, v_1) \in (s \cup s') : v_1 > v\}\big) \end{aligned}$$

with $\mathtt{max_k}(s)$ a function that returns the $k$ largest elements $(n, v) \in s$, with the elements ordered using the total order defined previously.

In general, this approach can be used to create CRDTs that compute a filter over the values added to the object, for which an element that does not match the filter at some moment will not match the filter at a later moment.

## 5. Design 3: Partially Incremental Computations

We now consider computations that are only partially incremental, in the sense that some updates observe the incremental property previously defined, while others do not. An example of such an object is a top-K object where an element can be deleted. In such cases, a value that does not belong to the top-K elements may later become part of the top, after a top element is deleted.

To address this case, a possible approach is to use a Set CRDT to maintain the set of elements that have not been deleted. In this case, all replicas maintain the complete set, and all updates need to be propagated to all replicas. The top-K can be computed locally on the value of each replica.

In Figure 3 we present an alternative approach, in which each replica maintains all operations locally executed, and each replica only propagates to other replicas the operations that might affect the computed result. Each replica maintains a set of operations and the results of the computation performed at other sites — for simplicity of notation, we assume that the result of the computation is a subset of operations. An update operation updates the local set of operations. A read operation makes the computation considering the local operations and the results of the computation at the other replicas. For synchronizing replicas, a replica sends the results of the computations to all replicas and the subset of operations known locally that can affect the computed result at other replicas (in the top-k example, a delete of an element that belongs to the top elements). When receiving the state from a remote replica, the local replica is updated by merging the local set of operations with the remote operations that may affect the result of the computation, and by registering the most recent version of the computation for each site.

A top-k replicated data type that supports an $\mathtt{add}(n, v)$ and $\mathtt{del}(n)$ operations can be defined as follows:

$$\begin{aligned} V_0 &= \{\} \\ \mathtt{fun}(s) &= \mathtt{max_k}\big(\{o \in s : o = \mathtt{add}(n, v) \wedge \\ &\qquad (\nexists o' \in s : o \prec o' \wedge o' = \mathtt{del}(n))\}\big) \end{aligned}$$

with $\mathtt{max_k}(s)$ a function that returns the $k$ $\mathtt{add}(n, v)$ operations with largest values $(n, v)$ for different values of $n$ and elements ordered using the total order defined previously. $\mathtt{fun}^{\mathtt{max}}$ can be defined by assigning a monotonic integer to the result computed in each replica, and using this integer to decide which value is the most recent.

| | | | |
|---|---|---|---|
| Replica state | $\Sigma$ | $=$ | $\big(\mathcal{P}(\mathtt{op}), \mathbb{I} \to V\big)$ |
| Initial state | $\sigma_i^0$ | $=$ | $\big(\{\}, i \to V_0\big)$ |
| Update op at replica $i$ | $\mathtt{op}_i\big((s,m)\big)$ | $=$ | $\big(s \cup \{\mathtt{op}\}, m\big)$ |
| Read at replica $i$ | $\mathtt{op}_i\big((s,m)\big)$ | $=$ | $\mathtt{fun}\big(s \bigcup_{\forall j} m[j]\big)$ |
| State to send from replica $i$ | $\mathtt{diff}\big(s,m\big)$ | $=$ | $(\{o \in s : \mathtt{fun}\big(o \bigcup_{\forall j} m[j]\big) \neq \mathtt{fun}\big(\bigcup_{\forall j} m[j]\big)\}, m[i \to \mathtt{fun}(s \bigcup_{\forall j} m[j])])$ |
| Merge replica states | $\mathsf{deliver}\ \big((s,m),(s',m')\big)$ | $=$ | $\big(s \cup s', m[j \mapsto \mathtt{fun}^{\mathtt{max}}(m[j], m'[j])] \forall j\big)$ |

Figure 3: Generic replicated data type for partially incremental computation.

This design enforces eventual consistency, assuming that replicas continue synchronizing until they reach an equivalent state, i.e., a state where read operations return the same result in every replica. However, this may not happen after the first synchronization step. For example, consider a top-1 object replicated in two sites: Site 1 executed operations $\{\mathtt{add}(b,15), \mathtt{add}(a,10)\}$ and site 2 executed operations $\{\mathtt{add}(b,16), \mathtt{add}(c,12)\}$. The two sites synchronize, with the top-1 element, $(b,16)$, being known at both replicas. After this, $\mathtt{del}(b)$ executes at site 1, promoting $(a,10)$ to the top at site 1. After the propagation of $\mathtt{del}(b)$ to site 2, $(c,12)$ is promoted to the top at site 2. After the next synchronization step, the top at site 1 $(a,10)$ is replaced by the same value as in site 2 $(c,12)$.

## 6. Final remarks

In this paper we have proposed three generic designs for replicated data types that perform a computation on the operations executed by users. These designs can be used in a system that maintains CRDT replicas at multiple sites and synchronizes them using a state-based model. We present the properties that computations must obey in order to use each of the designs. These designs try to minimize the information that each replica has to maintain and propagate to other replicas for synchronization.

The last proposed design departs from the strict CRDT state-based model, while still enforcing eventual consistency. We are currently formalizing the new model and studying the relations between replicated data types implemented using this design and state-based CRDTs that implement the same functionality. In the future, we intend to study how to integrate these designs in an eventually consistent cloud database, such as Riak.

## Acknowledgments

This research is supported in part by FCT scholarship SFRH / BD / 65070 / 2009, FCT projects PTDC/ EEI-SCR/ 1837/ 2012 and PEst-OE/ EEI/ UI0527/ 2014 and EU FP7 SyncFree project (609551).

## References

[1] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In *Proc. of the Third International Conference on Networked Systems (NETYS) (to appear)*, May 2015.

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

[3] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *Proc. VLDB Endow.*, 7(13):1441–1451, Aug. 2014.

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, 2014. ACM.

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

[6] 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 *Proc. of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007. ACM.

[7] 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 *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, 2011. ACM.

[8] C. Meiklejohn and P. Van Roy. Lasp: A Language for Distributed, Eventually Consistent Computations with CRDTs. In *Proc. of the Workshop on Principles and Practice of Consistency for Distributed Data*, Apr. 2015.

[9] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. Incremental stream processing using computational conflict-free replicated data types. In *Proc. of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, pages 31–36, 2013. ACM.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proc. of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, 2011. Springer-Verlag.

[11] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. *Comput. Netw.*, 52(12):2292–2330, Aug. 2008.

# F Swiftcloud: Write fast, Read in the past: Causal consistency for client-side applications

# Write Fast, Read in the Past: Causal Consistency for Client-side Applications

Marek Zawirski
Inria Paris-Rocquencourt &
Sorbonne Universités,
UPMC Univ Paris 06, LIP6

Nuno Preguiça
Sérgio Duarte
U. Nova de Lisboa

Annette Bieniusa
U. of Kaiserslautern

Valter Balegas
U. Nova de Lisboa

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

## Abstract

Client-side (e.g., mobile or in-browser) apps need local access to shared cloud data, but current technologies either do not provide fault-tolerant consistency guarantees, or do not scale to high numbers of unreliable and resource-poor clients, or both. Addressing this issue, we describe the SwiftCloud distributed object database, which supports high numbers of client-side partial replicas. SwiftCloud offers fast reads and writes from a causally-consistent client-side cache. It is scalable, thanks to small and bounded metadata, and available, tolerating faults and intermittent connectivity by switching between data centres. The price to pay is a modest amount of staleness. This paper present the SwiftCloud algorithms, design, and experimental evaluation, which shows that client-side apps enjoy the same guarantees as a cloud data store, at a small cost.

## 1. Introduction

Client-side applications, such as in-browser and mobile apps, are poorly supported by the current technology for sharing mutable data over the wide-area. App developers resort to ad-hoc application-level caching and buffering implementations, in order to avoid slow, costly and sometimes unavailable round-trips to a data centre, but they cannot solve system issues such as fault tolerance or session guarantees [41]. Recent application frameworks such as Google Drive Realtime API [17], TouchDevelop [15] or Mobius [18] support client-side access at a small scale, but do not provide system-wide consistency and/or fault tolerance guarantees. Algorithms for geo-replication [5, 7, 22, 30, 31] or for managing database replicas on clients [12, 33] ensure some of the right properties, but were not designed to support high numbers of client replicas.

Our thesis is that the system should be ensuring correct and scalable database access to client-side applications, addressing the (somewhat conflicting) requirements of consistency, availability, and convergence [32], at least as well as server-side systems. Concurrent updates (which are unavoidable if updates are to be always available) should not be lost, nor cause the database to diverge permanently. Under these requirements, the strongest consistency model is *causal consistency* with *convergent objects* (CRDTs) [30, 32, 39].

Supporting thousands or millions of client-side replicas challenges classical assumptions. To track causality precisely, per client, creates unacceptably fat metadata; but the more compact server-side metadata management has fault-tolerance issues. Additionally, full replication at high numbers of resource-poor devices would be unacceptable [12]; but partial replication of data and metadata could cause anomalous message delivery or unavailability. Furthermore, it is not possible to assume, like many previous systems [5, 22, 30, 31], that fault tolerance or consistency is solved, because the application is located inside the data centre (DC), or has a sticky session to a single DC [8, 41].

This work addresses these challenges. We present the algorithms, design, and evaluation of SwiftCloud, the first distributed object store designed for a high number of replicas. It efficiently ensures consistent, available, and convergent access to client nodes, tolerating failures. To enable both small metadata and fault tolerance, SwiftCloud uses a flexible client-server topology, and decouples reads from writes. The client *writes fast* into the local cache, and *reads in the past* (also fast) data that is consistent, but occasionally stale. Our approach involves two major techniques:

**Cloud-backed support for partial replicas** (Section 3) A DC serves a consistent view of the database to the client, which the client merges with its own updates. In some failure

situations, a client may connect to a DC that happens to be inconsistent with its previous DC. Because the client does not have a full replica, it cannot fix the issue on its own. We leverage "reading in the past" to avoid this situation in the common case, and provide control over the inherent trade-off between staleness and unavailability. A client observes a remote update only if it is stored in some number $K \geq 1$ of DCs [33]. The higher the value of $K$, the more likely that a $K$*-stable* version is in both DCs, but the higher the staleness.

**Protocols with decoupled, bounded metadata** (Section 4) Thanks to funnelling communication through DCs and to "reading in the past," SwiftCloud leverages decoupled metadata [28] separating *tracking causality*, which uses small vectors assigned in the background by DCs, from *unique identification*, based on client-assigned scalar timestamps. Consequently, the metadata is small and bounded in size. Furthermore, a DC can prune its log independently of clients, replacing it with a summary of delivered updates.

We implement SwiftCloud and demonstrate experimentally that our design reaches its objective, at a modest staleness cost. We evaluate SwiftCloud in Amazon EC2, against a port of WaltSocial [40] and against YCSB [19]. When data is cached, response time is two orders of magnitude lower than for server-based protocols with similar availability guarantees. With three DCs (servers), the system can accommodate thousands of client replicas. Metadata size does not depend on the number of clients, the number of failures, or the size of the database, and increases only slightly with the number of DCs: on average, 15 bytes of metadata overhead per update, compared to kilobytes for previous algorithms with similar safety guarantees. Throughput is comparable to server-side replication, and improved for high locality workloads. When a DC fails, its clients switch to a new DC in under 1000 ms, and remain consistent. Under normal conditions, 2-stability causes fewer than 1% stale reads.

The paper is organised as follows. In Section 2, we state the problem by motivating the system model and the requirements of client-side database access. In Section 3, we present the design principles of SwiftCloud, and its concrete protocols in Section 4. In Section 5 we present our experimental results, followed by discussion of related work in Section 6.

## 2. Problem overview

We consider support for a variety of client-side applications, sharing a database of **objects** that the client can `read` and `update`. We aim to scale to thousands of clients, spanning the whole internet, and to a database of arbitrary size.

Figure 1 illustrates our system model. A cloud infrastructure connects a small set (say, tens) of geo-replicated data centres, and a large set (thousands) of clients. A DC has abundant computational, storage and network resources. Similarly to Sovran et al. [40], we abstract a DC as a powerful sequential process that hosts a **full replica** of the database.[1] DCs communicate in a peer-to-peer way. A DC may fail (e.g., due to disaster, power outage, WAN partition or misconfiguration [6, 27]) and recover with its persistent memory intact.

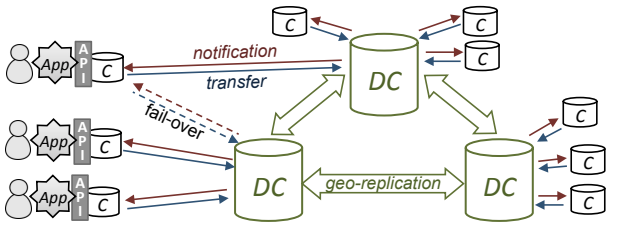



**Figure 1.** System components (Application processes, Clients, Data Centres), and their interfaces.

Clients do not communicate directly, but only via DCs. Normally, a client connects to a single DC; in case of failure or roaming, to zero or more. A client may fail and recover (e.g., disconnection during a flight) or permanently (e.g., destroyed phone) without prior warning. We consider only non-byzantine failures.[2]

Client-side apps require high **availability** and **responsiveness**, i.e., to be able to read and update data quickly and at all times. This can be achieved by replicating data locally, and by synchronising updates in the background. However, a client has limited resources; therefore, it hosts a **cache** that contains only the small subset of the database of current interest to the local app. It should not have to receive messages relative to objects that it does not currently replicate [37]. Finally, control messages and piggy-backed metadata should have small and bounded size.

Since a client replica is only *partial*, there cannot be a guarantee of complete availability. The best we can expect is **partial availability**, whereby an operation returns without remote communication if the requested data is cached; and after retrieving the data from a remote node (DC) otherwise. If the data is not there and the network is down, the operation may be unavailable, i.e., it either blocks or returns an error.

### 2.1 Consistency with convergence

Application programmers wish to observe a consistent view of the global database. However, with availability as a requirement, consistency options are limited [24, 32].

**Causal consistency** The strongest available and convergent model is causal consistency [3, 32].

DEFINITION 1 (Causal order, causal consistency). *Let an execution $E$ be a set of sequences (one per application*

[1] We refer to prior work for the somewhat orthogonal issues of parallelism and fault-tolerance within a DC [5, 22, 30, 31].

[2] Possible scalable approaches for Byzantine clients include admission control, and the Fork Join Causal Consistency protocol [33] to protect metadata, at the DC perimeter.

*process) of operation invocations with their return values. Operations* a *and* b *are* potentially causally-related *in $E$, noted* $a \rightarrow b$ *and called* ***causal order****:*

1. *An application process invoked* b *after it invoked* a.
2. *Read* b *observed update* a *on the same object.*
3. *There exists an operation* $c \in E$ *such that* $a \rightarrow c \rightarrow b$.

*An execution $E$ is* ***causally consistent*** *if every read operation in $E$ observes all the updates that causally precede the read, applied in some linear extension of causal order.*

Informally, under causal consistency, every process observes a *monotonically non-decreasing set of updates that includes its own updates, in an order that respects the causality between operations*.[3] The following well-known example illustrates [30]. In a social network, Bob sets permissions to disallow his boss Alice from viewing his photos. Some time later, Bob posts a questionable photo of himself. Without causal consistency, Alice may view the bad photo, delivered before the new permissions. Under causal consistency, the change of permission is guaranteed to be delivered before the post, and Alice cannot view the photo.

More generally, if an application process reads x, and later reads y, and if the state of x causally-depends on some update u to y, then the state of y that it reads will include update u. When the application requests y, we say there is a **causal gap** if the local replica has not yet received u. A consistent system must detect such a gap, and wait until u is delivered before returning y, or avoid it in the first place. If not, inconsistent reads expose both programmers and users to anomalies caused by gaps [30, 31].

We consider a transactional variant of causal consistency to facilitate multi-object operations: all the reads of a **causal transaction** come from a same database snapshot, and either all its updates are visible as a group, or none is [9, 30, 31].

**Convergence** Applications require **convergence**, which consists of liveness and safety properties: *(i)* **At-least-once delivery**: an update that is delivered (i.e., is visible by the app) at some node, is delivered to all interested nodes after a finite number of message exchanges; *(ii)* **Confluence**: nodes that delivered the same set of updates read the same value.

Causal consistency does not guarantee confluence, as two replicas might receive the same updates in different orders. For confluence, we rely on CRDTs, high-level data types with rich confluent semantics [16, 39]. An update on a high-level object is not just an assignment, but is a method associated with the object's type. For instance, a Set object supports add(element) and remove(element); a Counter supports increment() and decrement().

CRDTs include primitive last-writer-wins register (LWW) and multi-value register (MVR) [1, 21, 26], but also higher level types such as Sets, Lists, Maps, Graphs, Counters, etc. [2, 38–40]. Registers are simple to implement [22, 30, 31], but cumbersome to use. For instance, the implementation of LWW needs to store only the "last" update and has idempotent updates, but looses some concurrent assignments. Higher level types, such as Counter or Set, do not lose updates and are easier to use, but their implementation is more demanding. The implementation of high-level objects is eased by adequate support from the system. For instance, an object's value may be defined not just by the last update, but also depend on earlier updates; causal consistency is helpful, by ensuring that they are not lost or delivered out of order. However, safety also demands **at-most-once delivery**, as high-level updates are often not idempotent (consider for instance increment()).

Although each of these requirements may seem familiar or simple in isolation, the combination with scalability to high numbers of nodes and database size is a new challenge.

### 2.2 Metadata design

*Metadata* serves to identify updates and to ensure correct delivery. Metadata is piggy-backed on update messages, increasing the cost of communication.

One common metadata design assigns each update a timestamp as soon as it is generated on some originating node. The causality data structures tend to grow "fat." For instance, dependency lists [30] grow with the number of updates [22, 31, §3.3], whereas version vectors [12, 33] grow with the number of clients. (Indeed, our experiments hereafter show that their size becomes unreasonable). We call this the **Client-Assigned, Safe but Fat** approach.

An alternative delegates timestamping to a small number of DC servers [5, 22, 31]. This enables the use of small vectors, at the cost of losing some parallelism. However, this is not fault tolerant if the client does not reside in a DC failure domain. For instance, it may violate at-most-once delivery. Consider a client transmitting update u to be timestamped by DC1. If it does not receive an acknowledgement, it retries, say with DC2 (failover). This may result in u receiving two distinct timestamps, and being delivered twice. Duplicate delivery violates safety for many confluent types, or otherwise complicates their implementation [4, 16, 31]. We call this the **Server-Assigned, Lean but Unsafe** approach.

Clearly, neither "fat" nor "unsafe" is satisfactory.

### 2.3 Causal consistency with partial replication is hard

Since a partial replica receives only a subset of the updates, and hence of metadata, it could miss some causal dependencies [12]. Consider the following example: Alice posts a photo on her wall in a social network application (update a). Bob sees the photo and mentions in a message to Charles (update b), who in turn mentions it to David (update c). When David looks at Alice's wall, he expects to observe update a and view the photo. However, if David's machine

[3] This subsumes the well-known session guarantees [16].

does not cache Charles' inbox, it cannot observe the causal chain a → b → c and might incorrectly deliver c without a. Metadata design should protect from such causal gaps, caused by transitive dependency over absent objects.

Failures complicate the picture even more. Suppose David sees Alice's photo, and posts a comment to Alice's wall (update d). Now a failure occurs, and David's machine fails over to a new DC. Unfortunately, the new DC has not yet received Bob's update b, on which comment d causally depends. Therefore, it cannot deliver the comment, i.e., fulfill convergence, without violating causal consistency. David cannot read new objects from the DC for the same reason.[4]

Finally, a DC logs an individual update for only a limited amount of time, but clients may be unavailable for unlimited periods. Suppose that David's comment d is accepted by the DC, but David's machine disconnects before receiving the acknowledgement. Much later, after d has been executed and purged away, David's machine comes back, only to retry d. This could violate at-most-once delivery; some previous systems avoid this with fat version vectors [12, 33] or depend on client availability [28].

## 3. The SwiftCloud approach

We now describe an abstract design that addresses the above challenges, first in the failure-free case, and next, how we support DC failure. Our design demonstrates how to apply principles of causally consistent algorithms for full replication systems to build a cloud-based support for partial client replicas. Later, in Section 4, we present a concrete protocol implementing our design.

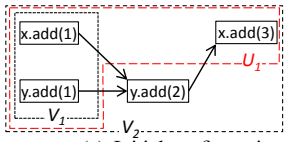
### 3.1 Causal consistency at full DC replicas

Ensuring causal consistency at fully-replicated DCs is a well-known problem [3, 22, 30, 31]. Our design is log-based, i.e., SwiftCloud stores updates in a log and transmits them incrementally; it includes optimisations, where the full log is occasionally replaced by the state of an object, called **checkpoint** [12, 35]. We discuss checkpoints only where relevant.

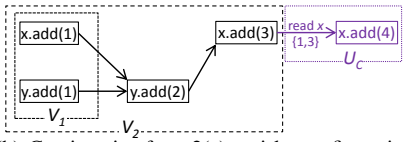
A **database version** is any subset of updates, noted $U$, ordered by causality. A version maps object identifiers to object state, by applying the relevant subsequence of the log; the value of an object is exposed via the read API.

We say that a version $U$ has a **causal gap**, or is **inconsistent** if it is not causally-closed, i.e., if $\exists \mathsf{u}, \mathsf{u}' : \mathsf{u} \rightarrow \mathsf{u}' \wedge \mathsf{u} \notin U \wedge \mathsf{u}' \in U$. As we illustrate shortly, reading from an inconsistent version should be avoided, because, otherwise, subsequent accesses might violate causality. On the other hand, waiting for the gap to be filled would increase latency and decrease availability. To side-step this conundrum, we adopt the approach of "reading in the past" [3, 30]. Thus, a DC exposes a gapless but possibly delayed state, noted $V$.

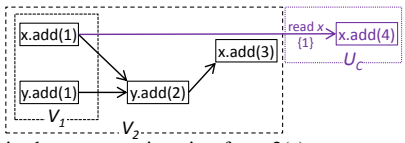
[4] Note that David can still perform updates, but they cannot be delivered, thus the system does not converge.



(a) Initial configuration



(b) Continuation from 2(a) to risky configuration



(c) Read-in-the-past: continuation from 2(a) to conservative configuration

**Figure 2.** Example evolution of configurations for two DCs, and a client. $x$ and $y$ are Sets; box = update; arrow = causal dependence (an optional text indicates the source of dependency); dashed box = named database version/state.

To illustrate, consider the example of Figure 2(a). Objects x and y are of type Set. $DC_1$ is in state $U_1$ that includes version $V_1 \subseteq U_1$, and $DC_2$ in a later state $V_2$. Versions $V_1$ with value $[\mathsf{x} \mapsto \{1\}, \mathsf{y} \mapsto \{1\}]$ and $V_2$ with value $[\mathsf{x} \mapsto \{1,3\}, \mathsf{y} \mapsto \{1,2\}]$ are both gapless. However, version $U_1$, with value $[\mathsf{x} \mapsto \{1,3\}, y \mapsto \{1\}]$ has a gap, missing update y.add(2). When a client requests to read x at $DC_1$ in state $U_1$, the DC could return the most recent version, $\mathsf{x} : \{1,3\}$. However, if the application later requests y, to return a safe value of y requires to wait for the missing update from $DC_2$. By "reading in the past" instead, the same replica exposes the older but gapless version $V_1$, reading $\mathsf{x} : \{1\}$. Then, the second read will be satisfied immediately with $\mathsf{y} : \{1\}$. Once the missing update is received from $DC_2$, $DC_1$ may advance from version $V_1$ to $V_2$.

A gapless algorithm maintains a *causally-consistent, monotonically non-decreasing progression* of replica states [3]. Given an update u, let us note $\mathsf{u}.deps$ its set of causal predecessors, called its **dependency set**. If a full replica, in some consistent state $V$, receives u, and its dependencies are satisfied, i.e., $\mathsf{u}.deps \subseteq V$, then it applies u. The new state is $V' = V \oplus \{\mathsf{u}\}$, where we note by $\oplus$ a **log merge operator** that filters out duplicates, further discussed in Section 4.1. State $V'$ is consistent, and monotonicity is respected, since $V \subseteq V'$.

If the dependencies are not met, the replica buffers u until the causal gap is filled.

### 3.2 Causal consistency at partial client replicas

As a client replica contains only part of the database and its metadata, this complicates consistency [12]. To avoid the complexity, we leverage the DC's full replicas to manage large part of gapless versions for the clients.

Given some **interest set** of objects the client is interested in, its initial state consists of the projection of a DC state onto the interest set. This is a causally-consistent state, as shown in the previous section. Client state can change either because of an update generated by the client itself, called an **internal update**, or because of one received from a DC, called **external**. An internal update obviously maintains causal consistency. If an external update arrives, without gaps, from the same DC as the previous one, it also maintains causal consistency.

More formally, consider some recent DC state, which we will call the **base version** of the client, noted $V_{DC}$. The interest set of client $C$ is noted $O \subseteq x, y, \ldots$. The client state, noted $V_C$, is restricted to these objects. It consists of two parts. One is the projection of base version $V_{DC}$ onto its interest set, noted $V_{DC}|_O$. The other is the log of internal updates, noted $U_C$. The client state is their merge $V_C = V_{DC}|_O \oplus U_C|_O$. On cache miss, the client adds the missing object to its interest set, and fetches the object from base version $V_{DC}$, thereby extending the projection.

Base version $V_{DC}$ is a monotonically non-decreasing causal version (it might be slightly behind the actual current state of the DC due to propagation delays). By induction, internal updates can causally depend only on internal updates, or on updates taken from the base version. Therefore, a hypothetical full version $V_{DC} \oplus U_C$ would be causally consistent. Its projection is equivalent to the client state: $(V_{DC} \oplus U_C)|_O = V_{DC}|_O \oplus U_C|_O = V_C$.

This approach ensures partial availability. If a version is in the cache, it is guaranteed causally consistent, although possibly slightly stale. If it misses in the cache, the DC returns a consistent version immediately. Furthermore, the client replica can *write fast*, because it does not wait to commit updates, but transfers them to its DC in the background.

Convergence is ensured, because the client's base version and log are synchronized with the DC in the background.

### 3.3 Failover: the issue with causal dependency

The approach described so far assumes that a client connects to a single DC. In fact, a client can switch to a new DC at any time, in particular in response to a failure. Although each DC's state is consistent, an update that is delivered to one is not necessarily delivered in the other (because geo-replication is asynchronous, to ensure availability and performance at the DC level [10]), which may create a causal gap in the client.

To illustrate the problem, return to the example of Figure 2(a). Consider two DCs: $DC_1$ is in (consistent) state $V_1$, and $DC_2$ in (consistent) state $V_2$; $DC_1$ does not include two recent updates of $V_2$. Client $C$, connected to $DC_2$, replicates object x only; its state is $V_2|_{\{x\}}$. Suppose that the client reads the Set x : $\{1, 3\}$, and performs update u = add(4), transitioning to the configuration shown in Figure 2(b).

If this client now fails over to $DC_1$, and the two DCs cannot communicate, the system is not live:

(1) *Reads are not available*: $DC_1$ cannot satisfy a request for y, since the version read by the client is newer than the $DC_1$ version, $V_2 \not\subseteq V_1$.
(2) *Updates cannot be delivered (divergence)*: $DC_1$ cannot deliver u, due to a missing dependency: $\mathsf{u}.deps \not\subseteq V_1$.

Therefore, $DC_1$ must reject the client (i.e., withhold his requests) to avoid creating the gap in state $V_1 \oplus U_C$.[5]

#### 3.3.1 Conservative read: possibly stale, but safe

To avoid such gaps that cannot be satisfied, the insight is to depend only on $K$**-stable updates** that are likely to be present in the failover DC, similarly to Mahajan et al. [33].

A version $V$ is $K$-stable if every one of its updates is replicated in at least $K$ DCs, i.e., $|\{i \in \mathcal{DC} \mid V \subseteq V_i\}| \geq K$, where $K \geq 1$ is a threshold configured w.r.t. expected failure model, and $\mathcal{DC}$ is a set of all data centers. To this effect, our system maintains a consistent $K$**-stable version** $V_i^K \subseteq V_i$, which contains the updates for which $DC_i$ has received acknowledgements from at least $K - 1$ distinct other DCs.

A client's base version must be $K$-stable, i.e., $V_C = V_{DC}^K|_O \oplus U_C|_O$, to support failover. In this way, the client depends, either on external updates that are likely to be found in any DC ($V_{DC}^K$), or internal ones, which the client can always transfer to the new DC ($U_C$).

To illustrate, let us return to Figure 2(a), and consider the conservative progression to Figure 2(c), assuming $K = 2$. The client's read of x returns the 2-stable version $\{1\}$, avoiding the dangerous dependency via an update on y. If $DC_2$ is unavailable, the client can fail over to $DC_1$, reading y and propagating its update remain both live.

By the same arguments as in Section 3.2, a DC version $V_{DC}^K$ is causally consistent and monotonically non-decreasing, and hence the client's version as well. Note that a client observes his internal updates immediately, even if not $K$-stable.

Parameter $K$ can be adjusted dynamically without impacting correctness. Decreasing it has immediate effect. Increasing $K$ has effect only for future updates, to preserve montonicity.

[5] The DC may accept to *store* client's updates to improve durability, but it cannot deliver them or offer notifications.

### 3.3.2 Causal consistency and partial replication: discussion

The source of the problem is an indirect causal dependency on an update that the two replicas do not both know about ($\mathsf{y.add}(2)$ in our example). As this is an inherent issue, we conjecture a general impossibility result, stating that genuine partial replication, causal consistency, partial availability and timely at-least-once delivery (convergence) are incompatible. Accordingly, the requirements must be relaxed.

Note that in many previous systems, this impossibility translates to a trade-off between consistency and availability on the one hand, and performance on the other [20, 30, 40] By "reading in the past," we displace this to a trade-off between freshness and availability, controlled by adjusting $K$. A higher $K$ increases availability, but updates take longer to be delivered;[6] in the limit, $K = N$ ensures complete availability, but no client can deliver a new update when some DC is unavailable.A lower $K$ improves freshness, but increases the probability that a client will not be able to fail over, and that it will block until its original DC recovers. In the limit, $K = 1$ is identical to the basic protocol from Section 3.2, and is similar to blocking session-guarantee protocols [41].

$K = 2$ is a good compromise for deployments with three or more DCs that covers common scenarios of a DC failure or disconnection [20, 27]. Our evaluation with $K = 2$ shows that it incurs a negligible staleness.

**Network partitions** Client failover between DCs is safe and generally live, except when the original set of $K$ DCs were partitioned away from both other DCs and the client, shortly after they delivered a version to the client. In this case, the client blocks. To side-step this unavoidable possibility, we provide an unsafe API to read inconsistent data.

When a set of fewer than $K$ DCs is partitioned from other DCs, the clients that connect to them do not deliver their mutual updates until the partition heals. To improve liveness in this scenario, SwiftCloud supports two heuristics: *(i)* a partitioned DC announces its "isolated" status, automatically recommending clients to use another DC, and *(ii)* clients who cannot reach another DC that satisfies their dependencies can use the isolated DCs with $K$ temporarily lowered, risking unavailability if another DC fails.

**Precision vs. missing dependencies.** The probability of a client blocked due to an unsatisfied transitive causal dependency depends on many factors, such as workload- and deployment-specific ones. Representation of dependencies also contributes. SwiftCloud uses coarse-grained representation of dependencies, at the granularity of the complete base version used by the client. This may cause a *spurious* missing dependency, when a DC rejects a client because it misses some update that is not an actual dependence. Finer-grained dependency representation, such as in causality graphs [30], or resorting to application-provided explicit dependencies under a weaker variant of causal consistency [7], avoid some spurious dependencies at the expense of fatter metadata. However, the missing dependency issue remains under any dependency precision. Thus, our approach is to fundamentally minimize the chances of *any* missing dependency, both genuine and spurious.

[6] The increased number of concurrent updates that this causes is not a big problem, thanks to confluent types.

## 4. Implementation

We now describe a metadata and concrete protocols implementing the abstract design.

### 4.1 Timestamps, vectors and log merge

The SwiftCloud approach requires metadata: *(1)* to *uniquely identify an update*; *(2)* to *encode its causal dependencies*; *(3)* to *identify* and *compare versions*; *(4)* and to *identify all the updates of a transaction*. We now describe a type of metadata, which fulfils the requirements and has a low cost. It combines the strengths of the two approaches outlined in Section 2.3, and is both *lean and safe*.

A timestamp is a pair $(i, k) \in (\mathcal{DC} \cup \mathcal{C}) \times \mathbb{N}$, where $i$ identifies the node that assigned the timestamp (either a DC or a client) and $k$ is a sequence number. Similarly to the solution of Ladin et al. [28], our metadata assigned to some update $\mathsf{u}$ combines both: *(i)* a single **client-assigned timestamp** $\mathsf{u}.t_{\mathcal{C}}$ that uniquely identifies the update, and *(ii)* a set of zero or more **DC-assigned timestamps** $\mathsf{u}.T_{\mathcal{DC}}$. Before being delivered to a DC, the update has no DC timestamp; it has one thereafter; it may have more than one in case of DC failover. Nodes can refer to an update via any of its timestamps to tolerate failures. The updates in a transaction all have the same timestamp(s), to ensure all-or-nothing delivery [40].

We represent a version or a dependency as a **version vector** [34]. A vector is a partial map from node ID to integer, e.g., $VV = [DC_1 \mapsto 1, DC_2 \mapsto 2]$, which we interpret as a set of timestamps. For example, when $VV$ is used as a dependency for some update $\mathsf{u}$, it means that $\mathsf{u}$ causally depends on $\{(DC_1, 1), (DC_2, 1), (DC_2, 2)\}$. In SwiftCloud protocols, every vector has at most one client entry, and multiple DC entries; thus, its size is bounded by the number of DCs, limiting network overhead. In contrast to a dependence graph, a vector compactly represents transitive dependencies and can be evaluated locally by any node.

Formally, the timestamps represented by a vector $VV$ are given by a function $\mathcal{T}$:

$$\mathcal{T}(VV) = \{(i, k) \in \mathsf{dom}(VV) \times \mathbb{N} \mid k \leq VV(i)\}$$

Similarly, the version decoding function $\mathcal{V}$ of vector $VV$ on a state $U$ selects every update in $U$ that matches the vector

($\mathcal{V}$ is defined for states $U$ that cover all timestamps of $VV$):

$$\mathcal{V}(VV, U) = \{\mathsf{u} \in U \mid (\mathsf{u}.T_{\mathcal{DC}} \cup \{\mathsf{u}.t_{\mathcal{C}}\}) \cap \mathcal{T}(VV) \neq \emptyset\}$$

For the purpose of the decoding function $\mathcal{V}$, a given update can be flexibly referred through any of its timestamps. Moreover, $\mathcal{V}$ is stable with growing state $U$. This is useful to identify a version on a large state that undergoes concurrent log appends; formally, $\forall VV, U, U' : U \subset U' \wedge \mathcal{T}(VV) \subseteq \bigcup_{\mathsf{u} \in U} (\mathsf{u}.T_{\mathcal{DC}} \cup \{\mathsf{u}.t_{\mathcal{C}}\}) \implies \mathcal{V}(VV, U) = \mathcal{V}(VV, U')$.

The log merge operator $U_1 \oplus U_2$, which eliminates duplicates, is defined using client timestamps. Two updates $\mathsf{u}_1 \in U_1, \mathsf{u}_2 \in U_2$ are identical if $\mathsf{u}_1.t_{\mathcal{C}} = \mathsf{u}_2.t_{\mathcal{C}}$. The merge operator merges their DC timestamps into $\mathsf{u} \in U_1 \oplus U_2$, such that $\mathsf{u}.T_{\mathcal{DC}} = \mathsf{u}_1.T_{\mathcal{DC}} \cup \mathsf{u}_2.T_{\mathcal{DC}}$.

### 4.2 Protocols

We now describe the protocols of SwiftCloud by following the lifetime of an update, and with reference to the names in Figure 1.

**State** A DC replica maintains its state $U_{DC}$ in durable storage. The state respects causality and atomicity for each individual object, but due to internal concurrency, this may not be true across objects. Therefore, the DC also has a vector $VV_{DC}$ that identifies a safe, monotonically non-decreasing causal version in the local state, which we note $V_{DC} = \mathcal{V}(VV_{DC}, U_{DC})$. Initially, $U_{DC}$ contains no updates, and vector $VV_{DC}$ is zeroed.

A client replica stores the commit log of its own updates $U_C$, and the projection of the base version from the DC, restricted to its interest set $O$, $V_{DC}|_O$, as described previously in Section 3.2. It also stores a copy of vector $VV_{DC}$ that describes the base version.

**Client-side execution** When the application starts a transaction $\tau$ at client $C$, the client initialises it with an empty buffer of updates $\tau.U = \emptyset$ and a **snapshot vector** of the current base version $\tau.depsVV = VV_{DC}$; the DC can update the client's base version concurrently with the transaction execution. A read in transaction $\tau$ is answered from the version identified by the snapshot vector, merged with recent internal updates, $\tau.V = \mathcal{V}(\tau.depsVV, V_{DC}|_O) \oplus U_C|_O \oplus \tau.U$. If the requested object is not in the client's interest set, $x \notin O$, the clients extends its interest set, and returns the value once the DC updates the base version projection.

When the application issues internal update $\mathsf{u}$, it is appended to the transaction buffer $\tau.U \leftarrow \tau.U \oplus \{\mathsf{u}\}$, and included in any later read. To simplify the notation, we assume hereafter that a transaction performs at most one update.[7] The transaction commits locally at the client and never fails.

[7] This can be easily extended to multiple updates, by assigning the same timestamp to all the updates of the same transaction, ensuring the all-or-nothing property [40].

If the transaction made update $\mathsf{u} \in \tau.U$, the client replica commits it locally as follows: *(1)* assign it client timestamp $\mathsf{u}.t_{\mathcal{C}} = (C, k)$, where $k$ counts the number of updates at the client; *(2)* assign it a **dependency vector** initialised with the transaction snapshot vector $\mathsf{u}.depsVV = \tau.depsVV$; *(3)* append it to the commit log of local updates on stable storage $U_C \leftarrow U_C \oplus \{\mathsf{u}\}$. This terminates the transaction; the client can start a new one, which will observe the committed updates.

**Transfer protocol: Client to DC** The transfer protocol transmits committed updates from a client to its current DC, in the background. It repeatedly picks the first unacknowledged committed update $\mathsf{u}$ from the log. If any of $\mathsf{u}$'s internal dependencies has recently been assigned a DC timestamp, it merges this timestamp into the dependency vector. Then, the client sends a copy of $\mathsf{u}$ to its current DC. The client expects to receive an acknowledgement from the DC, containing the timestamp(s) $T$ that the DC assigned to update $\mathsf{u}$. If so, the client records the timestamps in the original update record $\mathsf{u}.T_{\mathcal{DC}} \leftarrow T$. In a failure-free case, $T$ is a singleton.

The client may now transfer the next update in the log.

A transfer request may fail for three reasons:

(a) Timeout: the DC is suspected unavailable; the client connects to another DC (failover) and repeats the protocol.
(b) The DC reports a *missing internal dependency*, i.e., it has not received some update of the client, as a result of a previous failover. The client recovers by marking as unacknowledged all internal updates starting from the oldest missing dependency, and restarting the transfer protocol from that point.
(c) The DC reports a *missing external dependency*; this is also an effect of failover. In this case, the client tries yet another DC. The approach from Section 3.3.1 avoids repeated failures.

Upon receiving update $\mathsf{u}$, the DC verifies if its dependencies are satisfied, i.e., if $\mathcal{T}(\mathsf{u}.depsVV) \subseteq \mathcal{T}(VV_{DC})$. (If this check fails, it reports an error to the client, indicating either case (b) or (c)). If the DC has not received this update previously, as determined by client timestamps, i.e., $\forall \mathsf{u}' \in U_{DC} : \mathsf{u}'.t_{\mathcal{C}} \neq \mathsf{u}.t_{\mathcal{C}}$, the DC does the following: *(1)* Assign it a DC timestamp $\mathsf{u}.T_{\mathcal{DC}} \leftarrow \{(DC, VV_{DC}(DC) + 1))\}$, *(2)* store it in its durable state $U_{DC} \oplus \{\mathsf{u}\}$, *(3)* make the update visible in the DC version $V_{DC}$, by incorporating its timestamp(s) into $VV_{DC}$. This last step makes $\mathsf{u}$ available to the geo-replication and notification protocols, described hereafter. If the update has been received before, the DC looks up its previously-assigned DC timestamps. In either case, the DC acknowledges the transfer to the client with the DC timestamp(s). Note that some of these steps can be parallelised between transfer requests received from different client replicas, e.g., using batched timestamp assignment.

**Geo-replication protocol: DC to DC** The geo-replication protocol relies on a uniform reliable broadcast across DCs. An update enters the geo-replication protocol when a DC accepts a fresh update during the transfer protocol. The accepting DC broadcasts it to all other DCs. The broadcast implementation stores an update in a replication log until every DC receives it. A DC that receives a broadcast message containing u does the following: (1) If the dependencies of u are not met, i.e., if $\mathcal{T}(\mathsf{u}.depsVV) \not\subseteq \mathcal{T}(VV_{DC})$, buffer it until they are; and (2) incorporate u into durable state $U_{DC} \oplus \{\mathsf{u}\}$ (if u is not fresh, the duplicate-resilient log merge safely unions all timestamps), and incorporate its timestamp(s) into the DC version vector $VV_{DC}$. This last step makes it available to the notification protocol. The $K$-stable version $V_{DC}^K$ is maintained similarly.

**Notification protocol: DC to Client** A DC maintains a best-effort notification session, over a FIFO channel, to each of its connected clients. The soft state of a session includes a copy of the client's interest set $O$ and the last known base version vector used by the client, $VV_{DC}{}'$. The DC accepts a new session only if its own state is consistent with the base version provided by the client, i.e., if $\mathcal{T}(VV_{DC}{}') \subseteq \mathcal{T}(VV_{DC})$. Otherwise, the client is redirected to another DC, since the DC would cause a causal gap with the client's state (the solution from Section 3.3.1 avoids repeated rejections).

The DC sends over each channel a causal stream of update notifications.[8] Notifications are batched according to either time or to rate [12]. A notification packet consists of a new base version vector $VV_{DC}$, and a log of all the updates $U_\Delta$ to the objects of the interest set, between the client's previous base vector $VV_{DC}{}'$ and the new one. Formally, $U_\delta = \{\mathsf{u} \in U_{DC}|_O \mid \mathsf{u}.T_{\mathcal{DC}} \cap (\mathcal{T}(VV_{DC}) \backslash \mathcal{T}(VV_{DC}{}')) \neq \emptyset\}$. The client applies the newly-received updates to its local state, described by the old base version: $V_{DC}|_O \leftarrow V_{DC}|_O \oplus U_\delta$, and assumes the new vector $VV_{DC}$. If any of received updates is a duplicate w.r.t. to the old version or to a local update, the log merge operator handles it safely. Note that transaction atomicity is preserved, since all updates of a transaction share a common timestamp, thus either all fit in a batch $U_\delta$ or none does.

When the client detects a broken channel, it reinitiates the session, possibly on a new DC.

The interest set can change dynamically. When an object is evicted from the cache, the notifications are lazily unsubscribed to save resources. When it is extended with object x, the DC responds with the current version of x, which includes all updates to x up to the base version vector. To avoid races, a notification includes a hash of the interest set, which the client checks.

[8] Alternatively, the client can ask for invalidations instead, trading responsiveness for lower bandwidth utilization and higher DC throughput.

### 4.3 Object checkpoints and log pruning

Update logs contribute to substantial storage and, to smaller extent, network costs. To avoid unbounded growth, **pruning protocol** periodically replaces the prefix of a log by a *checkpoint*. In the common case, a checkpoint is more compact than the corresponding log of updates; for instance, a log containing one thousand increments to a Counter object and their timestamps, can be replaced by a checkpoint containing just the number 1000, and a version vector.

#### 4.3.1 Log pruning in the DC

The log at a DC provides *(a)* protection from duplicated update delivery implemented with $\oplus$ operator, as explained earlier, and *(b)* the capability to compute different versions, for application processes reading at different causal times. A log entry for update u can be replaced with a checkpoint once all of its duplicates have been filtered out, and once u has been delivered to all interested application processes.

Precise evaluation of expendability condition would require access to the client replica states. In practice, we need to *prune aggressively*, but without violating correctness. In order to reduce the risk of pruning a version not yet delivered to an interested application (which could force it to restart an ongoing transaction), we prune only a **delayed version** $VV_{DC}^{\Delta}$, where $\Delta$ is a real-time delay [30, 31]. If this heuristic fails, the consequences are not fatal: an ongoing client transaction may need to restart and repeat prior reads or return inconsistent data if desired, but the committed updates are *never* aborted.

To avoid duplicates, we extend DC metadata as follows. $DC_i$ maintains an **at-most-once guard** $G_i : \mathcal{C} \to \mathbb{N}$, which records the sequence number of each client's last pruned update. The guard is local to and shared at a DC. Whenever the DC receives a transfer request or a geo-replication message for update u with client timestamp $(C, k)$ and cannot find it in its log, it checks the at-most-once guard $G_i(C)$ entry. If the DC recognises that the update is a duplicate of a pruned update ($G_i(C) \geq k$), it ignores the update, except that it advances version vector to include all of the u's DC timestamps; for a transfer request, the DC replies with a vector $VV_i$, which is an overapproximation of the (discarded) information about the exact set of u's DC timestamps.

The notification protocol also uses checkpoints. On a client cache miss, instead of a complete log, the DC sends an equivalent checkpoint of the object, together the client's guard entry, so that the client can merge it with his log safely.

#### 4.3.2 Pruning the client's log

Managing the log at a client is simpler. A client logs his own updates $U_C$, which may include updates to object that is currently out of his interest set. This enables the client to read its own updates, and to propagate them lazily to a DC

| | **YCSB** [19] | **SocialApp** [40] |
|---|---|---|
| Type of objects | LWW Map | Set, Counter, Register |
| Object payload | $10 \times 100$ bytes | variable |
| Read txns | read fields (A: 50% / B: 95%) | read wall† (80%) see friends (8%) |
| Update txns | update field (A: 50% / B:5%) | message (5%) post status (5%) add friend (2%) |
| Objects / txn | 1 (non-txnal) | 2–5 |
| Database size | 50,000 objects | 50,000 users (400,000 objects) |
| Object popularity | uniform / Zipfian | uniform |
| Session locality | 40% (low) / 80% (high) | |

† *Read wall* is an update if page view statistics are enabled.

**Table 1.** Characteristics of applications/workloads.

when connected and convenient. An update u can be discarded as soon as it appears in $K$-stable base version $V_{DC}^K$, i.e., when the client becomes dependent on the presence of u at a DC. The client discards the corresponding updates: $U_C \leftarrow U_C \setminus \mathcal{V}(VV_i^K, U_C)$.

## 5. Evaluation

We implement SwiftCloud and evaluate it experimentally, in comparison to alternatives. We show that SwiftCloud serves: *(i)* fast response, under 1 ms for both reads and writes to cached objects (Section 5.3); *(ii)* throughput scalability of with the number of DCs, and support for thousands of clients with small metadata size, linear in the number of DCs (Section 5.4); *(iii)* fault-tolerance w.r.t. client churn (Section 5.5) and DC outages (Section 5.6); and *(iv)* low staleness, under 3% of stale reads (Section 5.7).

### 5.1 Implementation and applications

SwiftCloud and the benchmark applications are implemented in Java.[9] SwiftCloud uses an extendable library of CRDT types [39, op-based], in-memory storage,[10] Kryo for data marshalling, and a custom RPC implementation. A client cache has a fixed size and uses an LRU eviction policy. More elaborate approaches, such as object prefetching [12], are feasible.

Our client API resembles both production object stores, such as Riak 2.0 or Redis [2, 36], and prototype causal transactional stores, such as COPS or Eiger [30, 31]:[11]

| | |
|---|---|
| `begin_transaction()` | `read(object) : value` |
| `commit_transaction()` | `update(object, method(args...))` |

The actual API also includes caching options omitted here.

[9] https://github.com/SyncFree/SwiftCloud

[10] Our prototype can use BerkeleyDB for durable storage, but it was turned off in the present experiments.

[11] Unlike COPS or Eiger, we consider interactive transactions, i.e., accessed objects do not need to be predefined.

Along the lines of previous studies of causally-consistent systems [5, 7, 31, 40], we use two different benchmarks, YCSB and SocialApp, summarized in Table 1.

YCSB [19] serves as a kind of micro-benchmark, with simple requirements, measuring baseline costs and specific system properties in isolation. It has a simple key-field-value object model, implemented as a LWW Map type, using a default payload size of ten fields of 100 bytes each. YCSB issues single-object reads and writes. We use two of the standard YCSB workloads: update-heavy Workload A, and read-dominated Workload B. The object access pattern can be set to either uniform or Zipfian. YCSB does not rely on transactional semantics or high-level data types.

SocialApp is a social network application modelled after WaltSocial [40].[12] It employs high-level data types such as Sets, for friends and posts, LWW Register for profile information, Counter for counting profile visits, and object references. Many SocialApp objects grow in size over time (e.g., sets of posts). We are not concerned about this growth; a recent work of Briquemont [14] demonstrates how to implement object sharding in a SwiftCloud-like system. SocialApp accesses multiple objects in a causal transaction to ensure that operations such as reading a wall page and profile information behave consistently. Percentage in parantheses indicate the frequency of each operation in the workload. The SocialApp workload is read-dominated, but the ostensibly read-only operation of visiting a wall actually increments the wall visit counter when statistics are enabled, in metadata experiments. The user popularity distribution is uniform.

We use a 50,000-user database for both applications, except for smaller 10,000 users database for metadata experiments, to increase the stability of measurements.

We are not aware of any realistic benchmark for large-scale client-side replication designed for thousands of clients that would define a workload with long client sessions. We evaluate the system with short client sessions issuing more frequent operations than we expect in realistic workloads. The system behaviour under such a condensed workload is our proxy for the behaviour with more clients running slower sessions of longer duration.

In order to model the locality behaviour of a client, both YCSB and SocialApp are augmented with a facility to control access locality, mimicking social network access patterns [13]. Within a client session, a workload generator draws uniformly from a pool of session-specific objects with either 40% (*low locality*) or 80% (*high locality*) probability. For SocialApp, the pool contains objects of user's friends. Objects not drawn from this local pool are drawn from the global (uniform or Zipfian) distribution described above. The local pool can fit in the client's cache.

[12] SocialApp does not implement WaltSocial's user registration operation, as it would that would require additional support for strong consistency.

### 5.2 Experimental setup

We run three DCs in geographically distributed Amazon EC2 availability zones (Europe, Virginia, and Oregon), and a pool of distributed clients. Round-Trip Times (RTTs) between nodes are as follows:

| | Oregon DC | Virginia DC | Europe DC |
|---|---|---|---|
| nearby clients | 60–80 ms | 60–80 ms | 60–80 ms |
| Europe DC | 177 ms | 80 ms | |
| Virginia DC | 60 ms | | |

Each DC runs on a single m3.m EC2 instance, cheap virtual hardware, equivalent to a single core 64-bit 2.0 GHz Intel Xeon processor (2 ECUs) with 3.75 GB of RAM, and OpenJDK7 on Linux 3.2. Objects are pruned at random intervals between 60–120 s, to avoid bursts of pruning activity. We deploy 500–2,500 clients on a separate pool of 90 m3.m EC2 instances. Clients load DCs uniformly and use the closest DC by default, with a client-DC RTT ranging in 60–80 ms.

For comparison, we provide three protocol modes: *(i) SwiftCloud mode* (default) with client cache replicas of 256 objects, and refreshed with notifications at a rate $\leq$1 s by default; *(ii) Safe But Fat metadata mode* with cache, but with client-assigned metadata only (modelled after PRACTI, or Depot without cryptography [12, 33]), *(iii) server-side replication mode* without client caches. In this mode, a read incurs one RTTs to a DC, whereas an update incurs two RTTs to a DC, modelling the cost of a synchronous write to a quorum of servers to ensure fault-tolerance comparable to SwiftCloud.

### 5.3 Response time and throughput

We run several experiments to compare SwiftCloud's client-side caching, with reference to the locality potential and server-side geo-replication without caching. For each workload we evaluate the system stimulated with different rates of aggregated incoming transactions, until it becomes saturated. We use a number of clients that is throughput-optimised for each pair of workload and protocol mode. We report aggregated statistics for all clients.

Figure 3 shows response times for YCSB, comparing server-only (left side) with client replication (right side), under low (top) and high locality (bottom), when the system is not overloaded. Recall that in server-only replication, a read incurs a RTT to the DC, whereas an update incurs 2 RTTs. We expect SwiftCloud to provide much faster response, at least for cached data. Indeed, the figure shows that a significant fraction of operations respond immediately in SwiftCloud mode, and this fraction tracks the locality of the workload (marked "locality potential" on the figure), within a $\pm$7.5 percentage-point margin attributable to caching policy artefacts.[13] The remaining operations require one round-trip to the DC, indicated as 1 RTT. As our measurements for

[13] A detailed analysis reveals the sources of this error margin. The default Zipfian object access distribution of YCSB increases the fraction of local accesses due to added "global" locality (up to 82,% local accesses for target 80% of workload session locality). On the other hand, low locality workload decreases amount of local accesses, due to magnified imperfections of LRU cache eviction algorithm (down to 34,% local accesses for target 40%).

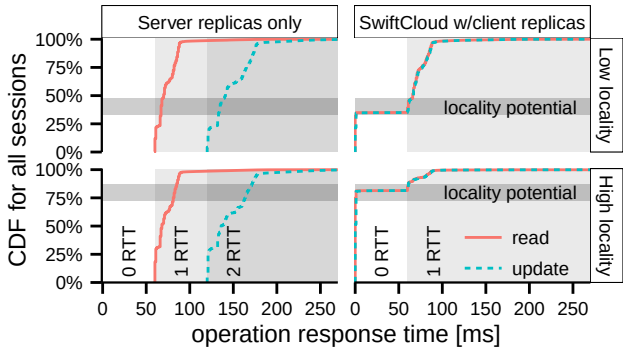



**Figure 3.** Response time for YCSB operations (workload A, Zipfian object popularity) under different system and workload locality configurations, aggregated for all clients.

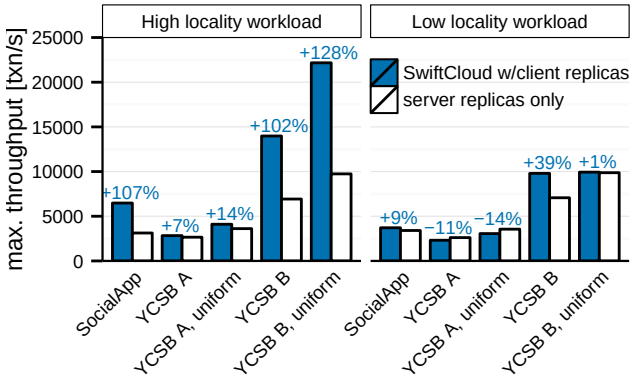



**Figure 4.** Maximum system throughput for different workloads and protocols. Percentage over bars indicates the increase/decrease in throughput for SwiftCloud compared to server-side replication.

SocialApp show the same message, we do not report them here. These results demonstrate that the consistency guarantees and the rich programming interface of SwiftCloud do *not* affect responsiveness of read and update of cached data.

In the next study, we saturate the system to determine its maximum aggregated throughput. Figure 4 compares SwiftCloud with server-side replication for all workloads.

Client-side replication is a mixed blessing: client replicas absorb read requests that would otherwise reach the DC, but on the other hand require additional work at the DC to maintain client replicas. The cost of client replicas pays off for read-dominated high locality workloads. SwiftCloud consistently delivers higher throughput for high locality workloads, by 7 % up to 128 %, and for read-heavy workloads in particular. In constrast, low locality workloads show no clear trend; depending on the workload, throughput either increases by up to 38 %, or decreases by to up 11 %.

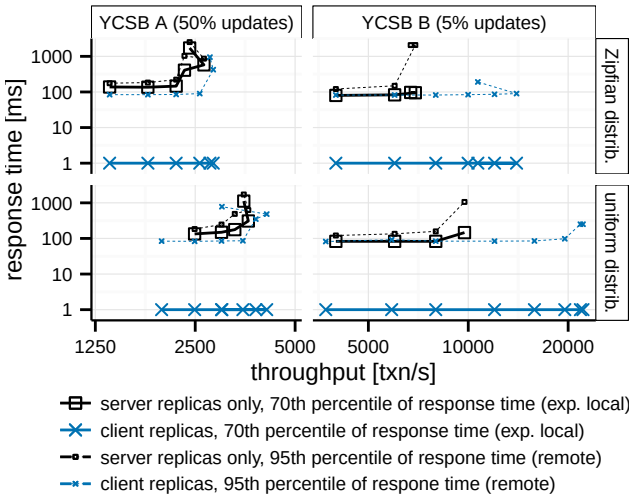



**Figure 5.** Throughput vs. response time for different system configurations running variants of YCSB.

Our next experiment studies how response times vary with server load and with the staleness settings. The results show that, as expected, cached objects respond immediately and are always available, but the responsiveness of cache misses depends on server load. For this study, Figure 5 plots throughput vs. response time, for YCSB A (left side) and B (right side), both for the Zipfian (top) and uniform (bottom) distributions. Each point represents the aggregated throughput and latency for a given transaction incoming rate, which we increase until reaching the saturation point. The curves report two percentiles of response time: the lower (70 th percentile) line represents the response time for requests that hit in the cache (the session locality level is 80%), whereas the higher (95 th percentile) line represents misses, i.e., requests served by a DC.

As expected, the lower (cached) percentile consistently outperforms the server-side baseline, for all workloads and transaction rates. A separate analysis, not reported in detail here, reveals that a saturated DC slows down its rate of notifications, increasing staleness, but this does not impact response time, as desired. In contrast, the higher percentile follows the trend of server-side replication response time, increasing remote access time.

Varying the target notification rate (not plotted) between 500 ms and 1000 ms, reveals the same trend: response time is not affected by the increased staleness. At a lower refresh rate, notification batches are less frequent but larger. This increases throughput for the update-heavy YCSB A (up to tens of percent points), but has no effect on the throughput of read-heavy YCSB B. We expect the impact of refresh rate to be amplified for workloads with smaller rate of notification updates.

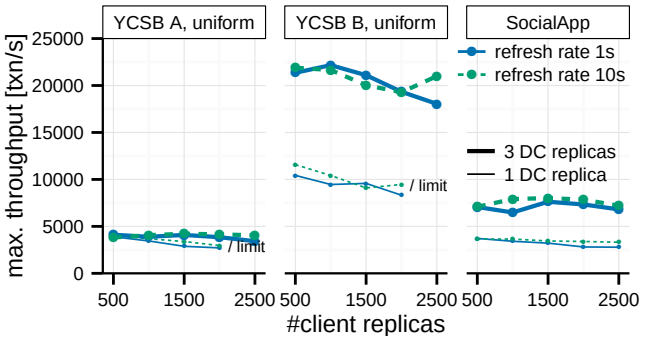



**Figure 6.** Maximum system throughput for a variable number of client and DC replicas.

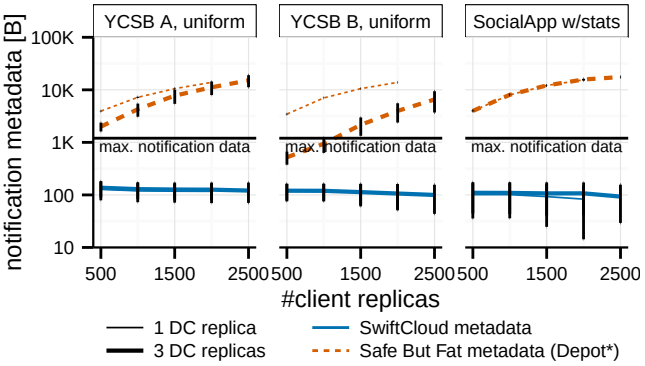



**Figure 7.** Size of metadata in notification message for a variable number of replicas, mean and standard error. Normalised to a notification of 10 updates.

### 5.4 Scalability

Next, we measure how well SwiftCloud scales with increasing numbers of DC and of client replicas. Of course, performance is expected to increase with more DCs, but most importantly, the size of metadata should be small, should increase only marginally with the number of DCs, and should not depend on the number of clients. Our results support these expectations.

In this experiment, we run SwiftCloud with a variable number of client (500–2500) and server (1–3) replicas. We report only on the uniform object distribution, because under the Zipfian distribution different numbers of clients skew the load differently, making any comparison meaningless. To control staleness, we run SwiftCloud with two different notification rates (every 1 s and every 10 s).

Figure 6 shows the maximum system throughput on the Y axis, increasing the number of replicas along the X axis. The thin lines are for a single DC, the bold ones for three DCs. Solid lines represent the fast notification rate, dashed lines the slow one. The figure shows, left to right, YCSB Workload A, YCSB Workload B, and SocialApp.

The capacity of a single DC in our hardware configuration peaks at 2,000 active client replicas for YCSB, and 2,500 for SocialApp. Beyond that, the DC drops sessions.

The wisdom of server-side replication applies in a new way to SwiftCloud: additional DC replicas increase the system capacity for operations that can be performed at only one replica. These traditionally include only read operations, but in the case of SwiftCloud, also sending notification messages (maintaining active clients). Whereas a single SwiftCloud DC supports at most 2,000 clients. With three DCs SwiftCloud supports at least 2,500 clients for all workloads. Unfortunately, as we ran out of resources for client machines at this point, we cannot report an upper bound.

For some fixed number of DCs, adding client replicas increases the aggregated system throughput, until a point of approximately 300–500 clients per DC, where the cost of maintaining client replicas up to date saturates the DCs, and further clients do not absorb enough reads to overcome that cost. Note that the lower refresh rate can reduce the load at a DC by 5 to 15%.

In the same experiment, Figure 7 presents the distribution of metadata size in notification messages. (Notifications are the most common and the most costly messages sent over the network.) We plot the size of metadata (in bytes) on the Y axis, varying the number of clients along the X axis. Left to right, the same workloads as in the previous figure. Thin lines are for one DC, thick lines for three DCs. A solid line represents SwiftCloud "Lean and Safe" metadata, and dotted lines the classical "Safe But Fat" approach. Note that our Safe-but-Fat implementation includes the optimisation of sending vector deltas rather than the full vector [33], as in Depot or PRACTI [12, 33] Vertical bars represent standard error across clients. As notifications are batched, we normalise metadata size to a message carrying exactly 10 updates, corresponding to under approx. 1 KB of data.

This plot confirms that the SwiftCloud metadata is small and constant, at 100–150 bytes/notification (10–15 bytes per update); data plus metadata together fit inside a single standard network packet.[14] It is *independent* both from the number of client replicas and from the workload, as well as from the number of objects in the database, as an additional experiment (not plotted) validates. Increasing the number of DC replicas from one to three causes a negligible increase in metadata size, of under 10 bytes. We attribute some variability to the data encoding and inaccuracies of measurements, including the normalisation process.

In contrast, the classical Safe-but-Fat metadata grows linearly with the number of clients and exhibits higher variability. Its size reaches approx. 1 KB for 1,000 clients in all workloads, and 10 KB for 2,500 clients. Clearly, metadata being up to $10\times$ larger than the actual data this represents a substantial overhead.

[14] The size of metadata does not exceed 100–150 bytes per notification with 10 updates, which matches our back of the envelope computations: 2–4 entries in a vector, plus 10 pairs of timestamps (one for each update) yields approximately 24 timestamps in total, with potential duplicates encoded more efficiently.

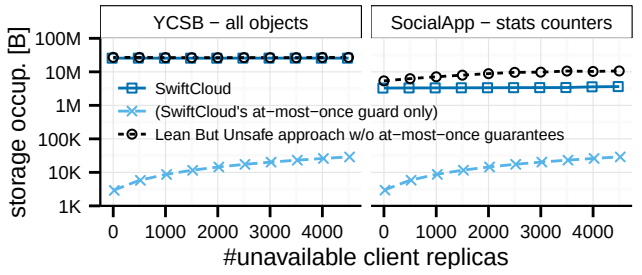



**Figure 8.** Storage occupation at one DC in reaction to client churn, for SwiftCloud and Lean-but-Unsafe alternative.

### 5.5 Tolerating client churn

We now turn to fault tolerance. In the next experiment, we evaluate SwiftCloud under client churn, by periodically disconnecting client replicas and replacing them with a new set of clients. At any point in time, there are 500 active clients and a variable number of disconnected clients, up to 5000. Figure 8 illustrates the storage occupation of a DC for representative workloads, which is also a proxy for the size of object checkpoints transferred. We compare SwiftCloud's log compaction to a protocol without at-most-once delivery guarantees (Lean But Unsafe).

SwiftCloud storage size is approximately constant thanks to the aggressive log compaction. This is safe thanks to the at-most-once guard table per DC. Although the size of the guard (bottom curve) grows with the number of clients, it requires orders of less storage than the actual database itself.

A protocol without at-most-once delivery guarantees uses Lean-but-Unsafe metadata, without SwiftCloud's at-most-once guard. However this requires more complexity in each object's implementation, to protect itself from duplicates. This increases the size of objects, impacting both storage and network costs. As is visible in the figure, the cost depends on the object type: none for YCSB's LWW-Map, which is naturally idempotent, vs. linear in the number of clients for SocialApp's Counter objects.

We conclude that the cost of maintaining SwiftCloud's at-most-once guard is negligible, and easily amortized by its stable behaviour and possible savings.

### 5.6 Tolerating DC failures

The next experiment studies the behaviour of SwiftCloud when a DC disconnects. The scatterplot in Figure 9 shows the response time of a SocialApp client application as the client switches between DCs. The client runs on a private machine outside of EC2. Each dot represents the response time of an individual transaction. Starting with a cold cache, response times quickly drops to near zero for transactions

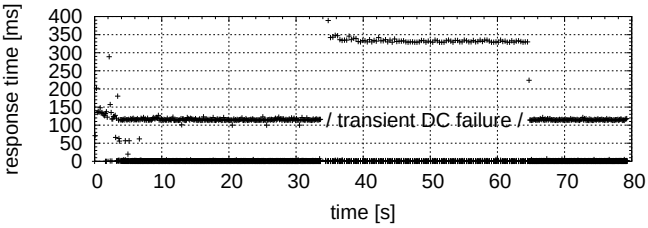



**Figure 9.** Response time for a client that hands over between DCs during a 30 s failure of a DC.

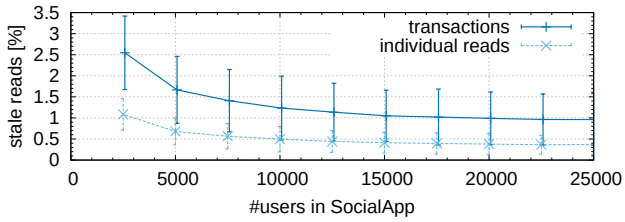



**Figure 10.** $K$-stability staleness overhead.

hitting in the cache, and to around 110 ms for misses. Some 33 s into the experiment, the current DC disconnects, and the client is diverted to another DC in a different continent. Thanks to $K$-stability the failover succeeds, and the client continues with the new DC. Response time for cache misses reflects the higher RTT to the new DC. At 64 s, the client switches back the initial DC, and performance smoothly recovers.

Recall that a server-side geo-replication system with similar fault-tolerance incurs high response time (cf. Section 5.3, or [20]) and does not ensure at-most-once delivery.

### 5.7 Staleness cost

The price to pay for our read-in-the-past approach is an increase in staleness. We consider a read *stale* if a version more recent (but not $K$-stable) than the one it returns exists at the current DC of a client that performed the read; a transaction is stale if any of its reads is stale. In the experiments so far, we observed a negligible number of stale reads and transactions, below 1 %. In another experiment, we artificially increase the probability of staleness by various means, e.g., using a smaller database, and setting cache size to zero. We run the SocialApp benchmark with 1000 clients in Europe connected to the Ireland DC and replicated in the Oregon DC.

Figure 10 shows that stale reads and stale transactions remain under $1\%$ and $2.5\%$ respectively. This shows that even under high contention, accessing a slightly stale snapshot has very little impact on the data read by transactions.

## 6. Related work

We now briefly discuss related results on consistency (Section 6.1), and compare similar systems and protocols in details (Section 6.2).

### 6.1 Consistency model and availability

Mahajan et al. [32] prove that no stronger consistency model than causal consistency is available and convergent, under *full* replication. We conjecture that these properties are not simultaneously achievable under *partial* replication, and we show how to weaken one of the liveness properties. Bailis et al. [8] also study variants of weak consistency models, and formulate a similar impossibility for a client switching server replicas. However, they do not take into account the capabilities of a client replica, leveraged by our solution.

Some operations or objects of application may require stronger consistency, which requires synchronous protocols [24]. For instance, we observe that our social network application port would benefit from strongly consistent support for user registration or a password change. Prior work demonstrates that combining strong and weak consistency is possible on shared data [29, 40]. Such techniques are applicable to SwiftCloud. Notably, Balegas et al. [11] propose protocols for preservation of consistency invariants as a middleware layer on top of SwiftCloud.

### 6.2 Existing systems

Several systems support consistent, available and convergent data access, at different scales. For the most related causally-consistent systems, we compare their metadata in Table 2. The columns indicate: *(i)* the extent of support for partial replication; *(ii)* which nodes assign timestamps; *(iii)* the worst-case size of causality metadata; *(iv)* the scope of validity of metadata representing a database version (is it valid only in a local replica or anywhere); *(v)* whether it ensures at-most-once delivery; *(vi)* whether it supports general confluent types (CRDTs).

#### 6.2.1 Replicated databases for client-side apps

PRACTI [12] is a seminal work on causal consistency under partial replication. PRACTI uses Safe-but-Fat client-assigned metadata and an ingenious log-exchange protocol that supports flexible communication topologies and options. While the generality of PRACTI has advantages, it is not viable for large-scale client-side replication deployment: *(i)* Its fat metadata approach (version vectors sized as the number of clients) is prohibitively expensive (see Figure 7), and *(ii)* any replica can easily make another unavailable, because of the indirect dependence issue discussed in Section 3.3.2. Our cloud-backed support of client-side replication addresses these issues at the cost of lower flexibility in communication topology. We are considering support for a limited form of peer-to-peer communication that would not cause these issues, e.g., between devices of a same user or a local group of collaborators.

Our high availability techniques are similar to Depot [33], a causally-consistent storage for client-side, built on top of untrusted cloud replicas. Depot tolerates Byzantine cloud

| | System | Partial replication support | Timestamp assignment | Causality metadata size $O$(#entries) | Version metadata validity | $\leq 1$ delivery | CRDTs support |
|---|---|---|---|---|---|---|---|
| **Safe but Fat** | PRACTI [12] | arbitrary, no sharding | any replica | #replicas $\approx 10^4$–$10^5$ | global | yes | possibly yes |
| | Depot [33] | partial data at client | any replica | #replicas $\approx 10^4$–$10^5$ | global | yes | possibly yes |
| | COPS-GT [30] | DC sharding | database client | \|causality subgraph\| | local (DC) | yes | possibly yes |
| | Bolt-on [7] | external sharding | DC server | \|explicit causality subgraph\| | local (DC) | no | LWW only |
| **Lean but Unsafe** | Eiger [31] | DC sharding | DC server (shard) | #objects $\approx 10^6$ | local (DC) | yes‡ | counter, LWW |
| | Orbe [22] | DC sharding | DC server (shard) | #servers $\approx 10^2$–$10^3$ | global† | no | LWW only |
| | ChainReaction [5] | DC sharding | DC (full replica) | #DCs $\approx 10^0$–$10^1$ | local (DC) | no | LWW only |
| | Walter [40] | arbitrary, no sharding | DC | #DCs $\approx 10^0$–$10^1$ | global | no | LWW only |
| **Lean and Safe** | Lazy Replication [28] | no partial replication | DC (full replica) + client | #DCs $\approx 10^0$–$10^1$ + 1 client entry | global | yes | single object |
| | SwiftCloud | no DC sharding, partial at client | DC (full replica) + client replica | #DCs $\approx 10^0$–$10^1$ + 1 client entry | global | yes | yes |

† not live during DC failures
‡ only for server-side replicas

**Table 2.** Qualitative comparison of metadata used by the most related causally consistent systems. "Timestamp assignment" indicates which nodes assign timestamps. "Causality metadata" indicates the type and maximum cardinality of entries in causality metadata, given as the expected order of magnitude (e.g., we expect hundreds to thousands servers). "Version metadata" indicates if a metadata to represent a consistent version is valid only in a local replica (DC), or globally; for the latter, if it is fault-tolerant or not. "$\leq 1$ delivery" indicates at-most-once delivery support.

behavior using cryptographic metadata signatures, in order to detect misbehavior, and fat metadata, in order to support direct client-to-client communication. Conservatively, Depot either exposes updates signed by $K$ different servers or forces clients to receive all transitive causal dependencies of their reads. This is at odds with genuine partial replication. Under no failures, a client receives metadata of *every* update; under failures, it may also receive their body. In contrast, SwiftCloud relies on DCs to compute $K$-stable consistent versions with lean metadata. In the event of an extensive failure involving $K$ DCs, SwiftCloud provides the flexibility to decrease $K$ dynamically or to weaken consistency.

Both Practi and Depot systems use Safe-but-Fat metadata, as indicated in Table 2. They support only LWW registers, but their rich metadata could conceivably accommodate high-level CRDTs too.

Lazy Replication (LR) protocols [28] support multiple consistency modes for client-side apps executing operations on server replicas. Under causal consistency, LR provides high availability with asynchronous read and write requests to multiple servers. As suggested by Ladin et al. [28], LR could also read stable updates for availability on failover, but that would force its clients to execute updates synchronously. The implementation of LR uses safe and lean metadata similar to SwiftCloud, involving client- and server-assigned timestamps together with version vector summaries. A log compaction protocol relies on client replicas availability and loosely-synchronised clocks for progress.

SwiftCloud structures the database into smaller CRDT objects, which allows it to provide partial client replicas, whereas LR considers only global operations. We show that client replicas can offer higher responsiveness on cached objects, instead of directing all operations to the server side as in LR, and that local updates can be combined with $K$-stable updates into a consistent view, avoiding slow and unavailable synchronous updates of LR. The log compaction technique of LR is complementary to ours, and optimises for the average case. SwiftCloud's aggressive pruning relies on at-most-once guard table, optimising for failure scenarios.

Recent web and mobile application frameworks, such as TouchDevelop [15], Google Drive Realtime API [17], or Mobius [18] support replication for in-browser or mobile applications. These systems are designed for small objects [17], database that fits on a mobile device [15], or a database of independent objects [18]. It is unknown if/how they support multiple DCs and fault tolerance. This is in contrast with SwiftCloud's support for large consistent database, and fault tolerance. TouchDevelop provides a form of object composition, and offers integration with strong consistency [15]. We are looking into ways of adapting similar mechanisms.

### 6.2.2 Geo-replicated databases for server-side systems

A number of geo-replicated systems offer available causally consistent data access inside a DC with excellent scale-out by sharding [5, 7, 22, 23, 30, 31].

Table 2 shows that server-side systems use variety of types of Lean-but-Unsafe metadata. COPS [30] assigns metadata directly at database clients, and uses explicit dependencies (a graph). Later work show that this approach is costly, and assigns metadata at object/shard replicas instead [22, 31], or on a designated node in the DC [5, 40]. The location of assignment directly impacts the size of causality metadata. In most systems, it varies with the number of reads, with the number of dependencies, and with the stability conditions in the system. When fewer nodes assign metadata, it tends to be smaller (as in SwiftCloud), but this may limit throughput. Recent work of Du et al. [23] make use of full stability, a special case of $K$-stability, to remove the need for dependency metadata in messages, thereby improving throughput.

Server-side designs do not easily extend beyond the scope and the failure domain of a DC, because *(i)* their protocols do not tolerate external client failures and DC outages, either blocking or violating safety (due to inadequate metadata, and the causal dependence issue); *(ii)* as they assume that data is updated by overwriting, implementing high-level confluent data types that work on the client-side is complex and costly (see Figure 8); *(iii)* their metadata can grow with database size.

SwiftCloud's support for sharding is limited compared to the most scalable decentralized server-side designs. Reconciling client-side replication with a more decentralized sharding support, and small metadata size, is future work. We believe this is possible to achieve by, once again, trading data freshness for performance, i.e., by managing a slightly stale consistent version at a high throughput, with small metadata [23].

## 7. Conclusion

We presented the design of SwiftCloud, the first object database that offers client-side apps a local access to partial replica with the guarantees of geo-replicated systems.

Our experiments show that the design of SwiftCloud is able to provide immediate and consistent response for reads and updates on local objects, and to maintain the throughput of a server-side geo-replication, or better. SwiftCloud's metadata allows it to scale safely to thousands of clients with 3 DCs, with small size objects, and metadata at the level of 15 bytes per update, independent of the number of connected and disconnected clients. Our fault-tolerant protocols handle failures nearly transparently, at a low staleness cost.

SwiftCloud's design leverages a common principle that helps to achieve several goals: client buffering and controlled staleness can absorb the cost of scalability, availability, and consistency.

Several aspects remain open for improvement and investigation. Our DC implementation is not sharded — we wish to combine modern sharded DC protocols with SwiftCloud, ideally without increasing the size of metadata. Practical applications require security mechanisms; we expect to adapt Depot's support for Byzantine clients, and additional access control and privacy mechanisms at the object level. We are also looking to better integration with programming model, in particular in terms of support for mixed weak and strong consistency, and for object composition [25].

**Acknowledgments** We would like to thank Carlos Baquero, Peter Bailis, Allen Clement, Alexey Gotsman, Masoud Saeida Ardekani, João Leitão, Vivien Quéma, and Luís Rodrigues for their comments that helped to improve this work. This research is supported in part by ANR (France) project ConcoRDanT (ANR-10-BLAN 0208), by Google Europe Fellowship in Distributed Computing 2010 awarded to Marek Zawirski, and by European FP7 project SyncFree (project 609 551, 2013–2016).

## References

[1] Riak, 2010. http://basho.com/riak/.

[2] Introducing Riak 2.0: Data types, strong consistency, full-text search, and much more, Oct. 2013. http://basho.com/introducing-riak-2-0/.

[3] M. Ahamad, J. E. Burns, P. W. Hutto, et al. Causal memory. In *Proc. 5th Int. Workshop on Distributed Algorithms*, pp. 9–30, Delphi, Greece, Oct. 1991.

[4] P. S. Almeida and C. Baquero. Scalable eventually consistent counters over unreliable networks. Number arXiv:1307.3207, July 2013.

[5] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on Chain Replication. In *Euro. Conf. on Comp. Sys. (EuroSys)*, Apr. 2013.

[6] P. Bailis and K. Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 2014.

[7] P. Bailis, A. Ghodsi, J. M. Hellerstein, et al. Bolt-on causal consistency. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pp. 761–772, New York, NY, USA, 2013.

[8] P. Bailis, A. Davidson, A. Fekete, et al. Highly Available Transactions: Virtues and limitations. In *Int. Conf. on Very Large Data Bases (VLDB)*, Riva del Garda, Trento, Italy, 2014.

[9] P. Bailis, A. Fekete, A. Ghodsi, et al. Scalable atomic visibility with RAMP transactions. In *ACM SIGMOD Conference*, 2014.

[10] P. Bailis, A. Fekete, M. J. Franklin, et al. Coordination avoidance in database systems. In *Int. Conf. on Very Large Data Bases (VLDB)*, Kohala Coast, Hawaii, 2015. To appear.

[11] V. Balegas, N. Preguiça, R. Rodrigues, et al. Putting the consistency back into eventual consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*, p. (To appear), Bordeaux, France, Apr. 2015.

[12] N. Belaramani, M. Dahlin, L. Gao, et al. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pp. 59–72, San Jose, CA, USA, May 2006.

[13] F. Benevenuto, T. Rodrigues, M. Cha, et al. Characterizing user behavior in online social networks. In *Internet Measurement Conference (IMC)*, 2009.

[14] I. Briquemont. Optimising Client-side Geo-replication with Partially Replicated Data Structures. Master's thesis, Universite Catholique de Louvain, Louvain-la-Neuve, Belgium, 2014.

[15] S. Burckhardt. Bringing TouchDevelop to the cloud. Inside Microsoft Research Blog, Oct. 2013. http://blogs.technet.com/b/inside_microsoft_research/archive/2013/10/28/bringing-touchdevelop-to-the-cloud.aspx.

[16] S. Burckhardt, A. Gotsman, H. Yang, et al. Replicated data types: Specification, verification, optimality. In *Symp. on Principles of Prog. Lang. (POPL)*, pp. 271–284, San Diego, CA, USA, Jan. 2014.

[17] B. Cairns. Build collaborative apps with Google Drive Realtime API. Google Apps Developers Blog, Mar. 2013. http://googleappsdeveloper.blogspot.com/2013/03/build-collaborative-apps-with-google.html.

[18] B.-G. Chun, C. Curino, R. Sears, et al. Mobius: Unified messaging and data serving for mobile apps. In *Int. Conf. on Mobile Sys., Apps. and Services (MobiSys)*, pp. 141–154, New York, NY, USA, 2012.

[19] B. F. Cooper, A. Silberstein, E. Tam, et al. Benchmarking cloud serving systems with YCSB. In *Symp. on Cloud Computing*, pp. 143–154, Indianapolis, IN, USA, 2010.

[20] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 251–264, Hollywood, CA, USA, Oct. 2012.

[21] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pp. 205–220, Stevenson, Washington, USA, Oct. 2007.

[22] J. Du, S. Elnikety, A. Roy, et al. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pp. 11:1–11:14, Santa Clara, CA, USA, Oct. 2013.

[23] J. Du, C. Iorgulescu, A. Roy, et al. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Symp. on Cloud Computing*, 2014.

[24] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700.

[25] A. Gotsman and H. Yang. Composite replicated data types. In *Euro. Symp. on Programming (ESOP)*, London, UK, 2015.

[26] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, Jan. 1976.

[27] A. Kansal, B. Urgaonkar, and S. Govindan. Using dark fiber to displace diesel generators. In *Hot Topics in Operating Systems*, Santa Ana Pueblo, NM, USA, 2013.

[28] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. *Operating Systems Review*, 25(1):49–55, Jan. 1991.

[29] C. Li, D. Porto, A. Clement, et al. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 265–278, Hollywood, CA, USA, Oct. 2012.

[30] W. Lloyd, M. J. Freedman, M. Kaminsky, et al. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 401–416, Cascais, Portugal, Oct. 2011.

[31] W. Lloyd, M. J. Freedman, M. Kaminsky, et al. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pp. 313–328, Lombard, IL, USA, Apr. 2013.

[32] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

[33] P. Mahajan, S. Setty, S. Lee, et al. Depot: Cloud storage with minimal trust. *Trans. on Computer Systems*, 29(4):12:1–12:38, Dec. 2011.

[34] J. Parker, D.S., G. J. Popek, G. Rudisin, et al. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Soft. Engin.*, SE-9(3):240–247, May 1983.

[35] K. Petersen, M. J. Spreitzer, D. B. Terry, et al. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 288–301, Saint Malo, Oct. 1997.

[36] Redis. Redis is an open source, BSD licensed, advanced key-value store. http://redis.io/, May 2014.

[37] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pp. 214–224, New Dehli, India, Oct. 2010.

[38] M. Shapiro, N. Preguiça, C. Baquero, et al. A comprehensive study of Convergent and Commutative Replicated Data Types. Number 7506, Rocquencourt, France, Jan. 2011.

[39] M. Shapiro, N. Preguiça, C. Baquero, et al. Conflict-free replicated data types. In *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pp. 386–400, Grenoble, France, Oct. 2011.

[40] Y. Sovran, R. Power, M. K. Aguilera, et al. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 385–400, Cascais, Portugal, Oct. 2011.

[41] D. B. Terry, A. J. Demers, K. Petersen, et al. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pp. 140–149, Austin, Texas, USA, Sept. 1994.

# G Technical report: Charcoal - A causally consistent protocol for geo-distributed partial replication


# Charcoal: A causally consistent protocol for geo-distributed partial replication

**Tyler Crain , Marc Shapiro**

**RESEARCH REPORT**

**N° 0000**

March 2015

Project-Teams Regal

ISSN 0249-6399 ISRN INRIA/RR--0000--FR+ENG


# Charcoal: A causally consistent protocol for geo-distributed partial replication

Tyler Crain *, Marc Shapiro *

Project-Teams Regal

Research Report n° 0000 — March 2015 — 17 pages

**Abstract:** Modern internet applications require scalability to millions of clients, response times in the tens of milliseconds, and availability in the presence of partitions, hardware faults and even disasters. To obtain these requirements, applications are usually geo-replicated across several data centres (DCs) spread throughout the world, providing clients with fast access to nearby DCs and fault-tolerance in case of a DC outage. Using multiple replicas also has disadvantages, not only does this incur extra storage, bandwidth and hardware costs, but programming these systems becomes more difficult.

To address the additional hardware costs, data is often *partially replicated*, meaning that only certain DCs will keep a copy of certain data, for example in a key-value store it may only store values corresponding to a portion of the keys. Additionally, to address the issue of programming these systems, consistency protocols are run on top ensuring different guarantees for the data, but as shown by the CAP theorem, strong consistency, availability, and partition tolerance cannot be ensured at the same time. For many applications availability is paramout, thus strong consistency is exchanged for weaker consistencies allowing concurrent writes like *causal consistency*. Unfortunately these protocols are not designed with partial replication in mind and either end up not supporting it or do so in an inefficient manner. In this work we propose a protocol designed to support partial replication under causal consistency more efficiently.

**Key-words:** distributed systems, causal consistency, partial replication

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 609551.

* Inria Paris-Rocquencourt & Sorbonne Universités, UPMC Univ Paris 06, LIP6

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

**Résumé :** Un protocole pour la géo-réplication, la cohérence causale, et la réplication partielle.

**Mots-clés :** systèmes distribués, la cohérence causale, la réplication partielle

# 1 Partial replication

Partial replication is becoming essential in geo-replicated systems to avoid spending uncescessary resources on storage and networking hardware. Implementing partial replication is more difficult than deciding how many replicas to have because protocols for data consistency must hide the organisation of replicas so that the programmer sees the data as a single continuous store. Furthermore ensuring consistency with partial replication does not always easily scale, as it often requires additional communication between nodes not involved in the operations. For example, in [8], Saeida shows that a scalable implementation of partial replication, namely one that ensures genuine partial replication [9] is not compatible with the snapshot-isolation consistency criterion. Differently, this work focuses on *causal consistency* which allows concurrent writes and uses meta-data propagation instead of synchronisation to ensure consistency, but even in this case, implementing partial replication in a scalable way is not a straightforward.

## 1.1 Causal consistency and partal replication

While protocols ensuring causal consistency are generally efficient when compared to strongly consistent ones, they often do not support partial replication by default or if they do, limit scalability by requiring coordination with nodes that do not replicate the values updated during propagation. Within the standard structure of these protocols, updates are performed locally, then propagated to all other replicas where they are applied respecting their *causal order*, which is given by session order and reads-from order or can be defined explicitly. Given the asynchrony of the system, propagated updates might arrive out of causal order at external replicas, thus before they are applied a *dependency check* must be performed to to ensure the correctness. This check is based on ordering meta-data that is propagated along with the updates or through separate messages [2].

The best known structure of this meta-data are vector-clocks where each totally-ordered participant is given a vector entry and each of their updates are assigned a unique increasing scalar value. Since these geo-replicated systems can have a large number of participants, they often use slightly different representations of the vector-clocks. For example, certain protocols use vectors with one entry per DC [13], or one entry per partition of keys [3], or use vectors that can be trimmed based on update stability or the organisation of the partitions of keys across DCs [3]. Other than vector clocks other approaches exist including real time clocks [5], or to track reads of memory locations [6] or operations [7] up to the the last update performed by this client.

Interestingly, all of these mechanisms create (over) approximations of the dependencies of each operation, i.e. from this meta-data you cannot tell exactly what the causal dependencies for this operation are, but for correctness they cover at minimum all the dependencies. For example when using a single vector entry per server, all operations from separate clients connected to this server will be totally ordered even if they access disjoint sets of data.

Such systems use these approximations primarily because precise tracking of dependencies would not scale as the size of the meta-data would grow up to the order of the number of objects or users in the systems (depending on how dependencies are tracked). The issue with over approximating dependencies is that the dependency check might have wait on more dependencies than necessary, allowing the client to read stale versions of the data. Fortunately though this does not block the progress of clients as updates are replicated outside of the critical path.

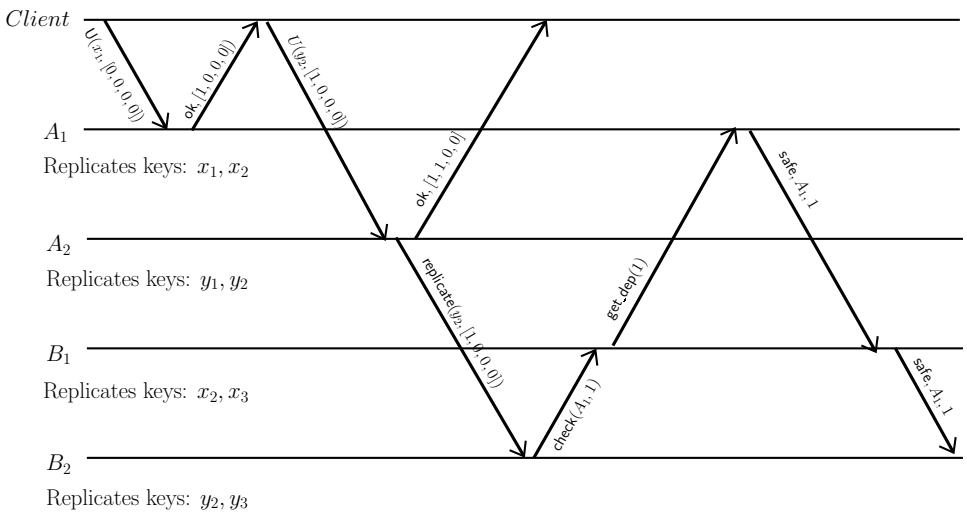



Figure 1: An example showing the possible dependency checks needed to ensure causality in a system with partial replication using version vectors with one entry per server for tracking dependencies.

## 1.2 Partial replication and approximate dependencies

This approximate tracking of dependencies also creates an unintended effect of making partial replication more costly.

To see why this happens, consider the example shown in figure 1 where a protocol is used that tracks dependencies using a version vector with an entry per server. Assume this protocol supports partial replication by only sending updates and meta-data to the servers that replicate the concerned object. In this example there are two DCs $DC_A$ and $DC_B$ each with 2 servers: $A_1$ and $A_2$ at $DC_A$ and $B_1$ and $B_2$ at $DC_B$. Server $A_1$ replicates objects $x_1$ and $x_2$, server $B_1$ replicates objects $x_2$ and $x_3$ while server $A_2$ replicates objects $y_1$ and $y_2$ and server $B_2$ replicates objects $y_2$ and $y_3$. The system starts in an initial state where no updates have performed. Consider then that a client performs an update $u1$ on object $x_1$ at server $A_1$, resulting in the client having a dependency vector of $[1, 0, 0, 0]$, with the 1 in the first entry of the vector representing the update $u1$ at $A_1$. Since $x_1$ is not replicated elsewhere the update stays locally at $A_1$. Following this, the client performs an update $u2$ on object $y_2$at server $A_2$, returning a dependency vector of $[1, 1, 0, 0]$. The update $u2$ is then propagated asynchronously to $B_2$, where upon arrival a dependency check is performed. Since the dependency vector inclues a dependency from $A_1$, before applying the update, $B_2$ must check with $B_1$ that it has received any updates covered by this dependency in case they were on a key replicated by $B_1$. But since $B_1$ has not heard from $A_1$ it does not know if the update was delayed in the network, or if the update involved an object it does not replicate. Thus $B_1$ must send a request to $A_1$ checking that is has received the necessary update. $A_1$ will then reply that it is safe because $u1$ did not modify an object replicated by $B_1$, which will then be forwarded to to $B_2$ at which time $u2$ can be safely applied. Notice that if dependencies were tracked precisely, this additional round of dependency checks would not be necessary as the dependency included with $u2$ would let the server know that it only depended on keys not replicated at $DC_B$.

While this is a simple example that one could imagine easily fixing, different workloads and topologies can create complex graphs of dependencies that are not so easily avoided. Furthermore, current protocols designed for full replication do not take any additional measures specifically to minimize this cost, instead they suggest to send the meta-data to every DC as if it was fully replicated either in a separate channel [2] or simply without the update payload. In effect using no specific design patterns to take advantage of partial replication.

# 2 Protocol Overview

The goal of this work then is to develop a protocol supporting partial replication and providing performance equal to fully replicated protocols in a full replication setting, while minimising dependency meta-data and checks in a partial replication setting. We will now give a short description of the main mechanisms used to design this algorithm. It should be noted that these mechanisms are common to many protocols supporting causal consistency, except here they are combined in a way with the goal of supporting partial replication.

- **Update identification** Vector clocks are the most common way to support causality. To avoid linear growth of vector clocks in the number of (client) replicas, we apply a similar technique as in the work of Zawirski et al.[13]. Each entry in a vector represents a DC, or more precisely a cluster within a DC. Modified versions of protocols such as ClockSI [4] or a DC- local service handing out logical timestamps, such as a version counter, can be used to induce a total order to the updates issued at this DC, which can then be represented in the DC's vector entry.

  For causality tracking, each update is associated with its unique timestamp given by its home DC, plus a vector clock describing its dependencies. To provide session guarantees such as causally consistent reads when interacting with clients, the client keeps a vector reflecting its previously observed values and writes. The system then ensures that clients may only read values containing all dependencies given this vector.

  Given that in partial replication a DC might not replicate all objects, certain reads will have to be forwarded to other DCs where the object being read is replicated. The receiving DC then uses the client's dependency vector to generate a consistent version of the object that is then forwarded to be cached at the DC the client is connected to.

- **Disjoint safe-time metadata** In general, most protocols ensure causal consistency by not making updates from external DCs visible locally to clients until all updates causally preceding it have been received. When objects are replicated at all DCs this is fairly straightforward as all dependent updates are expected to be received. This is not always the case in partial replication since only the replicated dependent operations should be received, which could result in additional messages or dependency checks (see figure 1 for an example of why these additional checks would be needed), something which we are trying to avoid in order to have an efficient implementation.

  To avoid these additional dependency checks and meta-data, the key insight in this work is to perform the dependency calculation at the origin DC and not the receiving DCs. Updates are still sent directly to their sibling replicas at other DCs, but they are not made visible to readers at the receiving DC until the origin DC confirms that its dependencies have been received i.e. the origin DC tracks which of its updates are safe to make visible at the receiver. At the origin DC, updates issued up to a time $t$ are considered safe to apply at a receiving DC when all of the origin DC's servers have sent all their updates on

the replicated objects of the receiver items up to time $t$. To keep track of this, a server at the origin DC communicates with each local server, keeping track of the time of the latest updates sent to external DCs, and once it has heard from each local server that time $t$ is safe, this information is then propagated to the external DCs as a single message. Doing this avoids unnecessary cross-DC dependency checks and meta-data propagation, saving computation and network bandwidth. The negative consequence of this is that the observable data at the receiving DC might be slightly more stale than in the full replication case because the receiving DC has to wait until the sending DC has let it know that this data is safe. Such a delay can be seen as a consequence of tracking dependencies approximately as seen in the example in figure 1

- **Local writes to non-replicated keys** Given that causal consistency allows for concurrent writes, in order to ensue low latency and high availability a DC will accept writes for all objects, including those that it does not replicate. Using the vector clocks and metadata as described above this can be done without any additional synchronisation by just assigning unique timestamps to these updates that are reflected in the vector of the local DC. These updates can then be safely logged and made durable even in the case of network partition.

- **Atomic writes and snapshot reads** Beyond simple key-value operations, the protocol provides a weak form of transactions which allows to group reads and updates together and supporting CRDT objects [10]. Atomic writes can be performed at the local DC using a 2-phase commit mechanism without contacting the remote replicas in order to allow for low latency and high availability. The updates are then propagated to the other DCs using the total ordered dependency metadata described previously ensuring their atomicity. (Note that atomic updates can include keys not replicated at the origin DC.) Causally consistent snapshot reads can be performed at a local DC by reading values according to a consistent vector clock, where reads of data items not replicated at the local DC are performed at another DC using the same vector clock.

Using these mechanisms allow partial or full replication with causal consistency while limiting the amount of unnecessary inter-DC meta-data traffic. All DCs are able to accept writes to any key, and causally consistent values can be read as long as one replica is available. Additionally the way the keys are partitions within a DC is transparent to external DCs, allowing this to be maintained locally. The following section presents the design of the protocol in detail.

# 3 Charcoal Protocol description

Charcoal is a modification of Antidote's [11] transaction protocol to allow for flexible replication and partitioning while still ensuring causal consistency. Specifically, there are two main differences from Antidote's original protocol: First is that instead of each DC having the same partition scheme for distributing keys between nodes, each DC can have a different partition scheme (the requirement still holds that their is at most one (primary) replica of each key-value pair per DC). Second is that each DC does not need to replicate the entire key space, thus allowing for partial replication. An interesting affect of these differences is that it allows a DC to be broken down into multiple groups, each separately running the full protocol and having their own entry in the vector. This property could be advantageous for example in a large DC with multiple clusters each partially replicating different portions of the key-space.

As in the full replication version, the transaction protocol here consists of two components.

- The first component handles the transaction management within a DC. These transactions are totally ordered using a modified version of the Clock-SI protocol [5], which assigns transactions growing scalar values using physical time-stamps creating the total order of transactions per DC. An important point of this protocol is that only the nodes of the partitions who replicate keys accessed by a transaction will participate in the transaction. This property, sometimes called *disjoint access parallelism* is important for scalability as it allows transactions who access disjoint portions of the key-space to execute in parallel. While the original protocol implements snapshot-isolation, which ensures a total order of writes with the DC, this modified version allows concurrent-writes as it is designed to work with CRDT objects and causal consistency. This is done by simply not validating the writes, thus avoiding unnecessary aborts and increasing concurrency.

- The second component is responsible for eventually (or asynchronously) replicating the updates from a transaction to other DCs. Both of them together guarantee that the transactions are causally consistent across all replicas of all objects.

The following data structures contain the meta-data required for causally consistent transactions.

**Meta-data per Partition**

- The *physical clock* $pc_j^k$ is the current physical clock of partition $k$ in DC $j$. It issues totally ordered time-stamps for this partition and is loosely synchronized with clocks for the other partitions in the same DC.

- Each partition has a queue *queue* that is stores incoming transactions as they are being received from other DCs.

- A vector $sentClock_j^k$ is kept at each partition storing time-stamps, with the number of entries equal to the number of DCs in the system. This vector keeps track of the most recent transaction time that this partition have sent to each external DC. For example a value of $sentClock_j^k[m] = 10$, means that partition $k$ at DC $j$ partition $l$ at DC $j$ has sent all messages up to time 10 to DC $m$.

- Each partition has a vector $safeToRead_j^k$, which stores time-stamps and same number of entries as the number of DCs. This vector keeps track of the time-stamps of updates from external DCs that are safe to be read by transactions executing at this DC. For example a value of $safeToRead_j^k[l] = 20$ means that all updates from DC $j$ up to time 20 have been received at DC $l$ and can be read by transactions. One thing that is important to note is that at any given time certain partitions within a DC may have smaller values in their $safeToRead$ vector than other partitions in the same DC, but given the way the protocol is designed, the maximum of these is actually safe at all partitions. This is important because if a transaction is given a snapshot with one of the larger values, it does not end up blocking in case another partition has not received this larger value yet.

- Each partition stores locally a map called the *partitoinTable* that maps a key to the DCs where this key is located. This is used so that a partition knows which DCs to send its update.

**Meta-data per Transaction**

- The vector snapshot time $vs$ denotes a snapshot derived at the starting point of the transaction. The transaction executes on this consistent snapshot identified by $vs$.

- The identifier $dc$ denotes the DC at which the transaction was originally executed and committed.

- The locations updated by a transaction are stored in its write set $WS$, including the key and modification done by the update operation.

- The commit time $c$ of a transaction is a vector clock with entries for each DC. If a transaction $T$ committed in DC $d$, then $T.c = C$ implies that $T$ has committed at time $C[d]$, where $C[d]$ is derived from physical clocks of the partitions involved in the transaction (see below). All other entries are taken from the vector snapshot time $vs$.

**Meta-data per Client**

- Each client keeps a client vector clock $cc$ with an entry corresponding to each DC. It keeps track of dependencies for client operations throughout a given session. All read and update operations should be executed on a snapshot with $cc \leq vs$. Normally this will be stored at an application server and will be given as input when starting a new transaction.

**Algorithm 1** Transaction coordinator $TC$ in partition $l$, DC $j$

**function** GetSnapshotTime(Clock $cc$)
  **for all** $i \leftarrow 0..D-1, i \neq j$ **do**
    $vs[i] \leftarrow \max(safeClock_j^l[i], cc[i])$
  **end for**
  $vs[j] \leftarrow \max(pc_j^l, cc[j])$
  **return** $vs$
**end function**

**function** StartTransaction(Transaction $T$, Clock $cc$)
  $T.vs \leftarrow$ GetSnapshotTime($cc$)
  $T.dc \leftarrow j$
  **return** $T$
**end function**

**function** Update(Transaction $T$, Key $k$, Operation $u$)
  $T.WS \leftarrow T.WS \cup \{\langle k, u\rangle\}$
**end function**

**function** Read(Transaction $T$, Key $k$)
  $\{\langle k, u\rangle\} \leftarrow T.WS \cap \{\langle k, \_\rangle\}$
  $\langle p, d\rangle \leftarrow$ partition($k$)
  $val \leftarrow$ send $\langle$ReadKey, $T.vs$, $k$, $u$, $j\rangle$ to $p$ at DC $d$
  **return** $val$
**end function**

**function** DistributedCommit($T$)
  **for all** $\langle K, U\rangle \in T$.WS **do**
    $\langle p\rangle \leftarrow$ partition($K$)
    **if** $p = \perp$ **then** $p \leftarrow l$
    **end if**
    $\langle T, U, p, d, timestamp\rangle \leftarrow$ send $\langle$Prepare, $T, K, U\rangle$ to $p$
    $T$.UpdatedPartitions $\leftarrow T$.UpdatedPartitions $\cup\{\langle p, d\rangle\}$
  **end for**
  CommitTime $\leftarrow \max($received $timestamps)$
  **if** CommitTime $< 0$ **then**
    $\langle T, \perp, p, d, timestamp\rangle \leftarrow$ send $\langle$Prepare, $T, \perp, \perp\rangle$ to $l$ at DC $j$
    $T$.UpdatedPartitions $\leftarrow T$.UpdatedPartitions $\cup\{\langle p, d\rangle\}$
    CommitTime $\leftarrow timestamp$
  **end if**
  $T.commitTime \leftarrow$ CommitTime
  **for all** $\langle p, d\rangle \in T$.UpdatedPartitions **do**
    send $\langle$Commit, $T\rangle$ to $p$ at DC $d$
  **end for**
  toPropagate$_j^l$.add($T$)
**end function**

## 3.1 Intra-DC transaction Protocol

The following described the protocol that is responsible for executing causally consistent transactions within a DC. It is a modified version of the fully replicated Antidote protocol and based on Clock-SI.

**Transaction Coordinator** A transaction coordinator whose behavior is described in Algorithm 1 is responsible for executing a transaction within a DC on behalf of clients. A client can contact any node in a DC and start a transaction coordinator $TC$. The client then issues update and read operations via $TC$.

**Start Transaction** When transaction $T$ starts, it is assigned a vector time-stamp $vs$, this vector is used as the causally consistent snapshot time for the reads performed by this transactions. This vector has as many entries as they are for DCs, with each entry describing which updates to read from that DC. When a key is read, all updates performed at each DC up to the time given by $vs$ have to be observed when performing the read operations of the transaction.

When assigning the values for this vector the client can provide a client clock $cc$, which is the last observed snapshot by the client. If a client clock $cc$ is provided, the transaction coordinator must assign a $vs$ that is at least as large as the provided values. Thus, using a client clock $cc$ guarantees that a client always observes monotonic snapshots even when connecting to other DCs.

Furthermore, values in this vector should be as large as possible to include the most recent updates, but not too big so that reads may have to block waiting for updates to be received. Thus, each location in the vector is assigned the maximum value for which all updates have been received from the corresponding external DC by using values from $safeClock$. The procedure for increasing $safeClock$ is part of the intra-DC replication protocol (described in section 3.2) and is shown in Algorithms 4 and 6. The $safeClock$ vector is increased when an external DC sends a message to the local DC letting it know that all updates up to the time have been received.

Since the clocks of partitions within the same DC are less likely to be out of sync for a long time we can assign $vs[j]$ (where $j$ is the DC where this transaction is being executed) to be the physical clock of the partition running the transaction coordinator, allowing $T$ to observed the latest committed transactions in DC $j$ This may then require a short amount waiting to occur at during operations of the transaction that are performed at other partitions in the same DC in case the clocks are out of sync, but safety is not violated. On the other hand, given that the protocol supports partial replication, reads to keys that are not replicated at the home DC must be performed at external DCs and use the same $vs$, meaning that the external DC must also have received all updates described by this vector. If there are frequent reads of keys that are not replicated at the transaction's home DC $vs[j]$ can be assigned a slightly older time to avoid waiting when performing these reads at external DCs.

After the transaction snapshot time is assigned, the client can issue read and update operations.

**Update** For update operations, the protocol simply logs the update locally in the write set $WS$ of the transaction.

**Read** The Read operation is called at the coordinator, which forwards a ReadKey request to the partition replicating the key requested. If the key has already been updated by the transaction (i.e. is in the write set), the update is returned directly. Otherwise, when performing the ReadKey operation, $T.vs$ is used to choose the version to read. This version is safe to read

**Algorithm 2** Transaction execution at partition $m$, DC $j$

```
1: function ReadKey(Transaction T, Key K, Update U, DC fromDC)
2:     if a transaction in the prepared list has a smaller time then wait
3:     if fromDC = j then
4:         return snapshot(K, T.vs, U)
5:     else
6:         Wait until all deps satisfied for fromDC, upto T.vs[fromDC]
7:         return snapshot(K, T.vs, U)
8:     end if
9: end function
10:
11: function Prepare(Transaction T, Key K, Update U)
12:     log U
13:     ⟨T2, prepareTime ⟩ ← preparedTransactions_j^m.get(T)
14:     if T2 = ⊥ then
15:         prepareTime ← pc_j^m
16:         if T.dc ≠ j then prepareTime ← ⊥
17:         end if
18:         preparedTransactions_j^m.add(T, prepareTime)
19:     end if
20:     send ⟨T, U, m, j, prepareTime⟩ to T's coordinator
21: end function
22:
23: function Commit(transaction T)
24:     log (commit, T.c, T.vs, T.commitTime)
25:     preparedTransactions_j^m.remove(T)
26: end function
```

because it was assigned using the $safeTime$ described in section 3.2, which guarantees that all partitions have received all updates from external DCs required by $T$'s snapshot. The read operation then only has to check if the updates from $T's$ home DC $j$ required for the snapshot are available in the partition. To ensure this, the protocol waits for the physical clock of the partition to increase past $T.vs[j]$ any for transactions who are in the process of committing with a prepare time smaller than than $T.vs[j]$ to commit.

Given that the local DC might not contain all keys that are read by this transaction, a read might need to be performed at an external DC. In this case the main difference is that at the external DC, before the snapshot of the object to be read is generated, the protocol has to ensure that all updates from the transaction's home DC have been received at the external DC (including all of the external dependencies from these updates). Notice that in the case of a network partition, some of the external dependencies may never be received thus causing the transaction to block, but this a consequence of not having full replication.

**Distributed Commit** When a transaction has finished, the commit operation is called, which executes in two rounds or phases at the local DC to ensure the atomicity of updates, following this the updates are sent asynchronously to the external DCs who replicate any of the keys updated by the transaction. During the first phase, the coordinator requests prepare timestamps from partitions involved in the transaction. It does this by going through the write set of the transaction, sending prepare requests containing the update operations of the transactions

to the partitions with the key being modified.

Because of partial replication, certain keys updated by the transaction might not be replicated by the local DC. In this case, instead of involving an external DC that does replicate the key, the prepare for these keys is performed at the coordinator. This is done in order to to ensure the high availability and low latency of update operations. In order to ensure the update is not lost it is still logged at the coordinator, but can be garbage collected as soon as it is propagated to all replicas.

Each of the partitions involved in the prepare add the transaction to the local queue $preparedTransactions$ and return the current physical time at the node. Once the coordinator has received all the times returned by the partitions involved in prepare it assigns maximum of these values as the commit-time of the transaction. This time is then sent to all participating partitions who then log it along with the updates. The two rounds ensure that the updates are atomic within the local DC because all reads of a transaction use the same $vs$ and reads are delayed in case a commit is in progress that might be assigned a smaller time. Finally the transaction is added to the *toPropagate* queue at the coordinator. The intra-DC replication part of the protocol is then responsible for propagating this transaction to other DCs asynchronously.

## 3.2 Intra-DC Replication Protocol

The intra-DC replication protocol is responsible for replicating transactions committed in one DC to other DCs. Given the asynchronous nature of the system, ensuring updates are propagated to all their replicas is not enough to ensure casual consistency, instead each operation must be performed in the order given by its reads-from and client session order. In addition to causality, we introduce following two additional requirements that are needed specifically for our high-availability geo-replicated system before describing the replication protocol in detail. The additional requirements are as follows:

- **Don't block waiting for dependencies** Consider that an update $u1$ is received at a DC and is immediately read by a client $c$, now $u1$ might causally depend on an update $u0$ to key $k$ which has not yet been received by $d$, so if $c$ wants to then read $k$ it must wait until $u0$ arrives. Since a primary goal of causal consistency is high availability, making clients wait until an update is received must be avoided, thus the first requirement is to ensure all causal dependencies have arrived before making a propagated update observable.

- **Include all previous updates** For certain applications it is enough that an update overwrites a previous one for example when using last-writer-wins (LWW). In such systems often only the value with the latest time-stamp is kept and other writes with smaller time-stamps are discarded. For this work we want to avoid automatically discarding any data as many applications prefer to keep all updates, which is sometimes needed for certain implementations of objects using causal+ consistency and is necessary for the implementation of several CRDTs. Thus when a value is read, all updates ordered causally before the read must have been received at the server performing the read.

The above requirements are ensured by the following condition

- A transaction $T$ from $p_i^m$ is applied (made visible to reading transactions) in $p_j^n$ $(i \neq j)$, only if $T$'s causal dependencies are satisfied locally in $p_j^m$. To ensure this, an update is only made visible when all updates from DC $j$ with time less than or equal to the commit time of the transaction have been received at DC $i$. This is done by having DC $j$ wait to let DC $i$ know these updates are available until the partitions of DC $j$ know that they have sent all preceding updates to DC $i$.

The following section will describe the intra-DC replication protocol in detail.

### 3.2.1 Protocol operations

**Algorithm 3** Replication Algorithm at the sender, running in partition m at DC i

```
1: function ReplicateToDC(j)
2:     loop
3:         time ← pc_j^m
4:         timePrepared ← max prepared time in preparedTransactions_j^m
5:         time ← max(time, timePrepared)
6:         Transactions ← toPropagate_j^m.removeAll()
7:         for all parallel T in Transactions do
8:             for all parallel D in DCs do
9:                 if T.ws ∩ replicationSet(D) ≠ ∅ then
10:                    send T to D
11:                    wait for ACK
12:                    if timeout then
13:                        sentToDC[D] ← false
14:                        toPropagate_j^m.add(T)
15:                    end if
16:                end if
17:            end for
18:        end for
19:        for all D in DC where sentToDC[D] = true do
20:            sentClock[D] ← time
21:        end for
22:    end loop
23: end function
```

The sending protocol consists of two procedures, ReplicateToDC described in Algorithm 3 which runs on each partition and is responsible for sending the transactions and SendSafeTime described in Algorithm 6 which runs on a single server per DC and is responsible for informing the receiving DC when it can safely make visible updates to read up to a given time (i.e. meaning that all updates ordered before this time at the sending DC have been received).

The receiver protocol also consists of two procedures. ReceiveTransaction described in Algorithm 4 which is run at each partition and receives transactions from external DCs, logs them, and in case the transaction updates other partitions in the DC, forwards them to those partitions. The second procedure, InformSafeExternal, runs on a single server per DC and receives safe times from the external DCs, letting the transaction protocol know when it is safe for local transactions to read updates that have been propagated from external DCs by updating the $safeTime$ vector.

**Replicating updates** Each partition runs a background process repeatably calling the ReplicateToDC procedure to send locally committed transactions to external DCs that replicate keys updated by the transactions. It starts by recording the minimum of the partitions physical clock and any transactions that are in the prepare phase of commit at this partition, any transaction that is not in the $to\_propagate_j^m$ at this point will be given a larger commit time than $time$. The protocol then goes through each transaction in the $to\_propagate_j^m$ queue, checking at which

DCs replicate its updates then sends to transaction to those DCs. Any DC that does not return an ACK for a sent transaction is recorded to keep track of which DCs did not receive all updates. For those DCs that did receive all updates, the *sent_clock* is updated, this vector has one entry per external DC and keeps track of the latest time for which all updates have been sent to the corresponding DC.

**Calculating the safe to read time** At each DC, there is a single server running that repeatably calls the SendSafeTime procedure. Each loop of this procedure requests the latest *sent_clock* vector from each partition in the DC. After collecting all these vectors, the procedure takes the minimum of these from each DC, and sends this value to the corresponding DC. This is the time at which the external DC has received all updates from this DC, meaning that the receiving DC can safely assign snapshots to new transactions that include updates up to this time.

**Algorithm 4** Replication Algorithm at the receiver, running in partition $m$ at DC $j$

```
1:  function ReceiveTransaction(Transaction T, DC i)
2:                          ▷ This function is repeatedly called to process transactions from DC i
3:      for all ⟨K, U⟩ ∈ T.WS do
4:          ⟨p, d⟩ ← partition(K)
5:          send ⟨LogTransaction, T, K⟩ to partition p at DC j
6:      end for
7:  end function
8:
9:  function LogTransaction(Transaction T, Key K)
10:     if T ∉ log then
11:         log T
12:     end if
13: end function
14:
15: function InformSafeExternal(DC d, Clock c)
16:     send ⟨InformSafeLocal, d, safeClock_j^m[d]⟩ to all partition in DC m
17: end function
18:
19: function InformSafeLocal(DC d, Clock c)
20:     safeClock_j^m[d] ← max(safeClock_j^m[d], c)
21: end function
```

**Processing received transactions** When a transaction is received from an external DC the ReceiveTransaction procedure is called. This procedure simply goes through the keys that are updated by the transaction, forwarding the transaction to the partitions that replicate them. When received at these partitions the transaction is logged. Note that when updates are then applied at the receiving DC, they cannot be read by new transactions until they have received all dependencies otherwise risking violating causal consistency. To prevent this, the receiving DC does not update the $safeTime$ until the previously received transactions have been safely logged. It is then the responsibility of the sending DC to let the receiver know when it is safe to read these times.

**Receiving safe times** As soon as a partition in DC $j$ receives a message from an external DC $i$ saying a time $c$ is safe in procedure InformSafeExternal, it means that all updates from $i$ up to this time have been received at $j$. This time $c$ is then gossiped around DC $i$ through InformSafeLocal to ensure all partitions know that updates with time equal to or smaller than $c$ from DC $i$ are safe to read without violating causality and can be included in future snapshots.

**Algorithm 5** Helper functions for keeping the local safe-time up to date at DC $m$ at DC $j$

```
1: function UpdateClocks
2:     if preparedTransactions_j^m ≠ ∅ then
3:         timestamps = Get prepare timestamps in preparedTransactions_j^m
4:         safeClock_j^m[j] ← min(timestamps) − 1
5:     else
6:         safeClock_j^m[j] ← pc_j^m
7:     end if
8: end function
9:
10: function UpdateSafe(DC d, Clock c)
11:     safeClock_j^m[d] ← max(safeClock_j^m[d], c)
12: end function
13:
14: function GetSafe
15:     reply ⟨safeClock_i^m[i]⟩
16: end function
```

**Algorithm 6** Sending safe times from DC $j$

```
1: function SendSafeTime
2:     loop
3:         sentClocks ← empty list
4:         for all parallel partition m in DC j do
5:             sentClock_m ← send ⟨GetSafeClock⟩ to m
6:             sentClocks.add(sentClock_m)
7:         end for
8:         for all parallel DC i in DCs do
9:             safeToAck_i ← min_{k∈partitions} sentClock_j^m[i][k]
10:            send ⟨InformSafeExternal, j, safeToAck_i⟩ to DC i
11:        end for
12:    end loop
13: end function
```

## 3.3 Dynamic Changing Replicas

Currently this document only contains descriptions of the transactional and intra-DC replication protocols. Future revisions will add the additional feature of allowing the partitioning and replication scheme during execution. It should be noted that the current protocol makes no specific requirements on how data is partitioned or replicated or not, thus adding these new

features will not largely change the protocols, but instead will focus on preventing data races when modifying the layout.

### 3.4 Implementation

An implementation of this protocol [12] is being developed within Antidote [11], the research platform for the SyncFree FP7 project, which is built on top of Riak-core [1] designed for testing scalable geo-replicated protocols.

## Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 609551. The Charcoal protocol builds upon the design and reuses many parts of the implementation of the Antidote protocol whose code is available at `https://github.com/SyncFree/antidote/` and whose contributors including Deepthi Akkoorath, Annette Bieniusa, Manuel Bravo, Zhongmiao Li, Christopher Meiklejohn, and Alejandro Zlatko Tomsic should be thanked for their help with developing Charcoal.

## References

[1] Basho. Riak-core. `https://github.com/basho/riak_core`, 2015.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Networked Sys. Design and Implem. (NSDI)*, pages 59–72, San Jose, CA, USA, May 2006. Usenix, Usenix. URL `https://www.usenix.org/legacy/event/nsdi06/tech/belaramani.html`.

[3] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Symp. on Cloud Computing*, pages 11:1–11:14, Santa Clara, CA, USA, Oct. 2013. Assoc. for Computing Machinery. doi: 10.1145/2523616.2523628. URL `http://doi.acm.org/10.1145/2523616.2523628`.

[4] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 173–184, Braga, Portugal, Oct. 2013. IEEE Comp. Society. doi: 10.1109/SRDS.2013.26. URL `http://doi.ieeecomputersociety.org/10.1109/SRDS.2013.26`.

[5] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Closing the performance gap between causal consistency and eventual consistency,. In *W. on the Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, the Netherlands, 2014. URL `http://eventos.fct.unl.pt/papec/pages/program`.

[6] 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 *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, Oct. 2011. Assoc. for Computing Machinery. doi: http://doi.acm.org/10.1145/2043556.2043593.

[7] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Networked Sys. Design and Implem. (NSDI)*, pages

313–328, Lombard, IL, USA, Apr. 2013. URL `https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final149.pdf`.

[8] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. PreguiÃ§a. On the scalability of snapshot isolation. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 369–381. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_39. URL `http://dx.doi.org/10.1007/978-3-642-40047-6_39`.

[9] N. Schiper, P. Sutra, and F. Pedone. P-Store: Genuine partial replication in wide area networks. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 214–224, New Dehli, India, Oct. 2010. IEEE Comp. Society. URL `http://doi.ieeecomputersociety.org/10.1109/SRDS.2010.32`.

[10] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. 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, Oct. 2011. Springer-Verlag. doi: 10.1007/978-3-642-24550-3_29. URL `http://www.springerlink.com/content/3rg3912287330370/`.

[11] SyncFree. Antidote reference platform. `https://github.com/SyncFree/antidote`, 2015.

[12] SyncFree. Antidote reference platform - partial replication branch. `https://github.com/SyncFree/antidote/tree/partial_replication`, 2015.

[13] 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. *arXiv preprint arXiv:1310.3107*, 2013.


**RESEARCH CENTRE**
**PARIS – ROCQUENCOURT**

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399