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

European Seventh Framework Program
ICT call 10

Deliverable reference number and title:	D.2.1 CRDTs at the server side
Due date of deliverable:	October 1, 2014
Actual submission date:	November 17, 2014
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:	CO

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 in 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 is architectures with replication at the server side. This scenario assumes a small number of replicas, a mostly-static topology, rare failures, and powerful servers.

The following paragraphs provide a short summary of our achievements in the first year. Detailed information is given in the subsequent chapters of this report.

Making operation-based CRDTs operation-based Conflict-free replicated data types (CRDTs) are the core concept of the SyncFree project. They can be classified as being either state-based or operation-based. Replicas of state-based CRDTs can be merged into a version where all updates performed against either replica are reflected. For operation-based CRDTs, only the updates are propagated and re-applied against the other replica. This reduces the bandwidth considerably, but requires the communication middleware to provide a reliable causal broadcast. Extending the foundational work of Shapiro et al. on CRDTs [40, 39], we improved on the operation-based CRDT design by reducing the complexity and meta-data involved. This work inspired us to build our research platform, Antidote, on operation-based CRDTS in contrast to state-of-the-art geo-replicated datastores such as Riak [13]. Details can be found in Section 4 and Appendix A.

Antidote In a collaboration involving most of the SyncFree academic partners and one industry partner, we developed our common experimental platform which provides the means for evaluating further research ideas and integrates results from all other WPs. Antidote is a geo-replicated CRDT data store which features scalable, conflict-free implementations of transactions, by providing consistent, stable snapshots and atomic multi-CRDT updates. Each data centre fully replicates the CRDT store using a static partitioning scheme. Updates are propagated as operations in a causally-consistent order on transactions, thus requiring a lower bandwidth than state-of-the-art architectures. Transactions with Causal Consistency+ semantics support the programmer in keeping data views consistent, while requiring no global synchronization. We introduce Antidote in detail in Section 5.

Adaptive replication Next, we have developed an algorithm for Adaptive Replication, together with tools to analyze and visualize Adaptive Replication. With the increasing growth of data that is accumulated and transferred to DCs for high accessibility and scalability, full replication for all data items becomes unfeasible eventually. The goal is therefore to find some 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. The basic idea is that accessing data items through reads and writes affects the likelihood with which the data will be replicated in a particular DC. To test a specific strategy, we provide a simulation tool to trace the paths of a data item. The tool visualizes how the shared object travels and replicates between the DCs under different configurations. More on adaptive replication can be found in Section 7, details on the simulation tool are available in the WP5 report.

SwiftCloud As a first contribution for supporting CRDTs at points-of-presence, we continued to develop and engineer our background platform, SwiftCloud. The objective of this work is to enable fault-tolerance and scalability by replicating data objects to clients. We developed protocols using full replication at server side and partial replication / sharding at client side. Our major contribution is the design for small and bounded meta-data and associated protocols to ensure scalability up to thousands of clients. Techniques such as object checkpoints and log pruning control the size of the object store. Access to locally cached objects reduces latency to a few milliseconds. Retrieving objects from a DC takes one round trip time in the normal case. In case of network partitions or DC failure, clients can re-connect safely to another DC while retaining a consistent data view. These advantages come at a low and adaptable cost in form of data staleness. A submitted article on SwiftCloud is provided in Section 6.

The results that we obtained from our work on SwiftCloud will be fed back into the development of our common experimental platform Antidote. In particular, the experiences on rendering a distributed data store faul-tolerant has inspired already some planned adaptations to Antidote's initial design (e.g. log layer and transaction management).

Link with other WPs The requirements for the initial version of the Antidote platform are taken from uses cases described in WP1. To validate the design decisions taken, we implemented the wallet application. This app will be used for starting extensive and comparative performance analysis in WP5.

The transaction algorithm for our experimental platform are building on our experience with transactions in SwiftCloud. They were developed in cooperation with WP3.

In addition to the development of algorithms and protocols, WP2 is also laying the foundation for integration of protocols and techniques from WP3 and WP4, as well as testing and profiling of the platforms to provide integration with WP5. The integration with other WPs will intensify in the coming months as the component reach maturity. For example, WP2 is working on integrating the Derflow programming model developed in WP4 as well as the bounded counter CRDTs and CRDT optimizations from WP3.

2 Milestones in the Deliverable

WP2.1 has reached the following milestone:

Mil. no	Mil. name	Date due	Actual date
MS1	CRDT consolidation in a static environment	M12	M12

The corresponding tasks are:

Task no	Task name	Date due	Actual date	Lead contractor
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

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. All milestones for WP2 have therefore been moved by 6 months. The executive board approved of this adjustment of the milestone 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 (together with WP4), Zhongmiao Li (together with WP4).

3.4 Nova

Valter Balegas (together with WP3), Nuno Preguica (together with WP3), Carlos Baquero (together with WP3).

3.5 Basho

Christopher Meiklejohn (together with WP4 and WP5).

3.6 Trifork

Amadeo Ascó (together with WP1).

4 Making Operation-based CRDTs Operation-based

In distributed databases, data replication can improve system performance and fault tolerance, but also impact the exposed level of data consistency. Offering the users the impression of an always-available single consistent copy is not easy in the presence of partitions among the replicas. As partitions, communication failures and topology changes are deemed to occur in all but the smallest systems, and since losing availability is normally not an option, developers have successfully explored relaxed consistency models [13], such as eventual consistency [46, 5].

In eventually consistent systems, data replicas are allowed to diverge; however, this divergence can be tracked so that, eventually, replicas can be reconciled into a common consistent state. In particular, causal consistency makes sure that each replica has access to all the operations that have influenced the state in the replica where an operation was first applied, before applying it locally.

Crafting, by hand, correct *merge* functions that can reconcile divergent replicas is costly and error prone, and errors can compromise eventual consistency. Merge functions depend on the particular semantics of the concrete datatype the replica is storing. For instance, in a replicated counter that is subject to *increment* operations, the objective of the *merge* would be to account for all distinct *increment* operations known to the replicas being merged. Conflict-free replicated datatypes (CRDTs) [39, 40] offer a model for designing correct replicated datatypes that are always-available and are guaranteed to eventually converge once all operations are known to all replicas.

CRDTs have two complementary designs:

- *Operation-based CRDTs* ship each received operation to all replicas, typically via reliable causal broadcast to ensure causal consistency. Replicas converge as long as causal dependencies are respected and the *effects* of concurrent operations are designed to be commutative, even if the operations are not commutative themselves.

- *State-based CRDTs* ship full state payloads, resulting from applying operations to a local replica state, and have a commutative, associative, and idempotent merge function that deterministically reconciles any two replica states. In mathematical terms, the merge in state-based CRDTs defines a least upper bound over a join-semilattice.

There is a trade-off between the above two approaches. Operation-based CRDTs can allow for simpler implementations and a simpler replica state, while requiring more guarantees from the message dissemination layer, namely, reliable causal broadcast. In contrast, state-based CRDTs require more complex states, i.e., storing more meta-data; however, they support ad-hoc dissemination of states, and can handle duplicate and out-of-order delivery of state payloads once merged at the destination replicas, without breaking causal consistency.

The current definition of operation-based CRDTs is very relaxed and allows for implementations that send extra information beyond to what is needed to identify an operation, e.g., sending sets of unique element identifiers when propagating a remove operation in an observed-removed set. This, makes it confusing to distinguish the difference between the two models, and imposes a notable source of inefficiency induced by this additional information.

In our work on operation-based CRDTs, we improve the current model of operation-based CRDTs by leveraging the causal meta-data already present in most reliable causal delivery broadcast protocols. The resulting model allows the exchange of small messages (only operation name and arguments) and a very compact state at the replicas. We call these CRDTs *pure operation-based.*

Example The observed-remove set (OR-set) is a set CRDT supporting both insertion and removal of elements. An element can only be removed if it had been observed at the respective node. The standard op-based implementation would look like this:

$$\begin{aligned}
\mathit{prepare}_i([\mathit{add}, v], (n, s)) &= [\mathit{add}, v, i, n+1] \\
\mathit{effect}_i([\mathit{add}, v, i', n'], (n, s)) &= (n' \text{ if } i = i' \text{ otherwise } n, s \cup \{(v, i', n')\}) \\
\mathit{prepare}_i[\mathit{rmv}, v], (n, s)) &= [\mathit{rmv}, \{(v', i', n') \in s | v' = v\}] \\
\mathit{effect}_i([\mathit{rmv}, r], (n, s)) &= (n, s \backslash r) \\
\mathit{eval}(\mathit{rd}, (n, s)) &= \{v | (v, i', n') \in s\}
\end{aligned}$$

Every update is split into two distinct operations. The *prepare* builds the message that is to be delivered to all replicas, whereas the *effect* actually applies the update based on the information gathered by prepare. For the OR-Set, the object state local to node i is represented as tuple of an operation counter n and value set s. The *add* operation prepares in addition to the value v to be added, a unique identifier $(i, n+1)$. When applying the effect, the counter is at the origin node increment, and the value is added to the set together with the unique identifier. When removing a value, the prepare collects all corresponding values in the set as concurrent adds can lead to duplicates. Finally, when evaluating the current value of the set, all meta-data (i.e. the unique identifiers) are stripped from the value set.

The pure op-based specification of the OR-set is considerably simpler:

$$\begin{aligned}
\mathit{prepare}(o, s) &= o \qquad \text{with } o \text{ either } [\mathit{add}, v] \text{ or } [\mathit{rmv}, v])\} \\
\mathit{effect}(o, t, s) &= s \cup \{(t, o)\} \\
\mathit{eval}(\mathit{rd}, s) &= \{v | (t, [\mathit{add}, v]) \in s \land \nexists (t', [\mathit{rmv}, v]) \in s \cdot t < t'\}
\end{aligned}$$

The object state is reduced to the value set, the prepare does not need to inspect the state of the object. However, the sender node generates a vector clock timestamp t to be delivered with the message. To fulfill the OR-set specification, the CRDT replica is represented as a partially ordered log of operations using the timestamp as index to the log. This gives a simple way of tracking causality between operations while reducing the meta-data involved. When evaluating the current value, data-type specific interpretations are applied as shown here for the OR-Set.

This work was presented at PaPEC in April 2014 and has been published in the proceedings of DAIS'14 [6]. The design of our experimental platform Antidote, as described in the next section, will soon apply the techniques of pure op-based CRDTs in one of its core components, namely the materializer cache (see Section 8).

5 Antidote

Distributed computing infrastructure is offered today by cloud services in data centres all around the world. Geo-scale applications and their underlying data stores can benefit from this infrastructure as it can reduce latency and availability when interacting with clients. To this end, these data stores rely heavily on replication of data, both within a data centre to tolerate server faults and to perform load balancing and load adaption, as well as across data centres to decrease latency and to increase reliability. As theoretical results and practical experience show, data replication in such a setting must aim for weaker notions of consistency than provided by classical strong synchronization schemes [14, 20, 38].

Conflict-free replicated data types (CRDTs) provide a principled approach to shared, mutable and replicated data under Eventual Consistency. An update can execute immediately, irrespective of network latency, faults, or disconnection at the local DC, and are then propagated to and replayed at the other DCs.

Though CRDTs relieve the programmer from constructing error-prone ad-hoc reconciliation solutions, they are not enough to guarantee application correctness. Usually, only updates to the same CRDT are observed in a causally consistent way by the client as there are no cross-object guarantees given.

In a joint effort involving KL, INRIA, UCL, Nova and Basho, we developed a common experimental platform targeting CRDT support at the server side with a small number of replicas in a controllable, confined, and stable execution environment such as a cloud. Antidote is a geo-replicated CRDT data store which features scalable, conflict-free implementations of transactions, by providing consistent, stable snapshots and atomic multi-CRDT updates. This type of transaction guarantees Causal Consistency+ semantics and follows herein other eventual consistent object stores.

Antidote is focusing on the following design objectives:

Low latency: Geo-replicating data on DC and replicating services within a DC, while caching objects and operations, reduces latency for read and write requests.

High scalability: Only carefully engineered protocols that rely on meta-data scaling with the number of DC as well as connected clients can provide scalable data stores. Further, mechanisms for garbage collection must be integrated to have systems running reliably over a long time.

High availability and fault-tolerance: Even when failures of machines or infrastructure occur, the system must be available and operate correctly. Fir example, updates should never be lost when a server crashes. Employing replication within DCs and making updates durable prevents this kind of information loss.

Robustness and safety: No internal information about the system's consistency mechanism is leaked to the client. Type checking and other safety mechanisms prevent the system from client-induced, unintended faults.

High-level programming abstractions: Programmers can use data types such as counters, sets, or maps. Causal Consistency and transactions help in reasoning about program invariants and data evolution.

Extensibility and adaptability: As a research platform, Antidote will continually evolve and must thus be extensible, adaptable and easily configurable.

The main rationale behind these design decisions is that the protocols and algorithms underlying Antidote need to be applicable in actual software systems under real-world conditions.

One of the design goals of Antidote is to have a layered architecture that enforces a clear separation of concerns, thus allowing to experiment with different approaches for specific tasks. For instance, the Transaction Layer is in charge of implementing protocols for retrieval and commit of objects and their update in accordance to the transaction semantics required.

The code for Antidote is available at

`http://github.com/SyncFree/antidote`,

including test cases, example applications and benchmarks. The repository also provides a detailed description for the installation process and software dependencies (see also Section 5.5.1). In Appendix C, the interfaces for the different components of Antidote's reference implementation is listed for documentation purpose.

Figure 1 shows the layered architecture of Antidote. In the following sections, we introduce the system setting, give a short introduction to each layer, and detail the behavior of the corresponding component. We provide for each layer at least on reference implementation. In future work, we will refine this layer system and provide a variety of implementations as well as integration of mechanisms defined in the work packages.

5.1 System setting

Antidote provides a multi-versioned key-value data store, where the data is stored as CRDTs. We assume to have a number of D stable data centres, where D is in the order of tens. At each DC, the data store is partitioned into P partitions. As for now, each DC employs the same static partitioning scheme, i.e., for every partition p_i^m in DC_i , there is a corresponding partition p_j^m in DC_j such that p_i^m and p_j^m replicate the same set of objects.

Every server is equipped with a hardware clock which is synchronized by a protocol such as the Network Time Protocol (NTP), such that the difference between clocks of different servers is bounded by the clock synchronization skew.

5.2 Log Layer

The log layer constitutes the foundation of the system architecture. Antidote uses a log-based backend to provide fast and fault-tolerant write access and efficient management of multi-versioning for CRDT objects. The log layer immediately accepts all append operations it receives. Different implementations of this component may write to disk, or to main memory of a quorum of processes, etc. Following the separation of concerns, any validation checks (e.g. for consistency or type correctness)

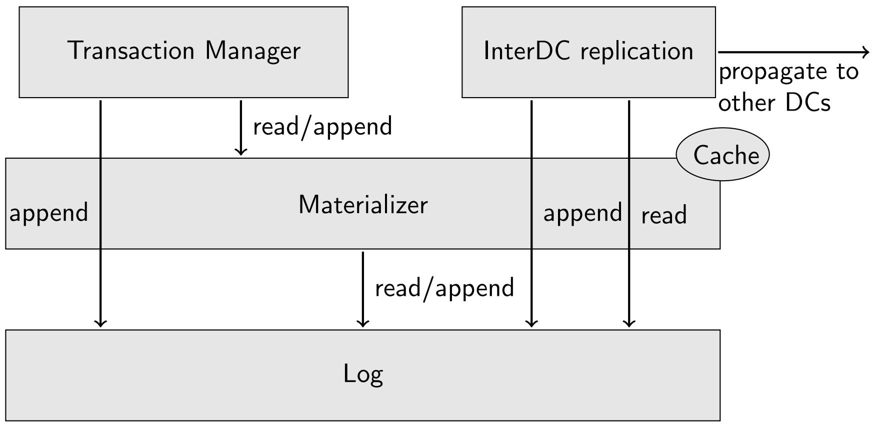


Figure 1: Architecture of Antidote.

are delegated to other layers. Similarly, generating object states by replaying the log is performed by a different layer.

Partioning a datastore can improve performance and scalability [18]. To simplify investigation of different partitioning schemes at a later point in time, Antidote implements a basic static datastore partitioning. For every DC, the logging layer is split into a number of partitions and each partition maintains its own log. A log is a sequence of operations. An operation is the composition of an operation type and an operation payload. The payload depends on the kind of operation performed, such as an update to some CRDT object or a transactional *abort* and *commit* markers. Besides the operation and potential parameters, the operation also contains the meta-data that the other layers require in order to obtain consistent and correct objects. The persistence component is oblivious of the actual operation and simply refers to an operation via a unique operation identifier.

The log layer's interface provides three operations:

- `append(partition, op)` inserts a new log entry for an operation `op` to the log of some partition;
- `read(partition)` returns all log entries for some partition;
- `read_from(partition, op_id)` returns the log entries for some partition that have been added after some op_id.

For these methods, the following guarantees are given:

- Once an append operation has been successfully acknowledged, it will always be contained in the operation log when reading from the partition.
- Once an operation has been returned in a read operation, it will always be contained in the operation log for subsequent reads.

Failure handling In the current version of the platform, we offer one basic logging mechanism: writing log entries synchronously to hard disk. This provides a simple, but not necessarily safe, fault-tolerance mechanism. In particular, Antidote does not replicate the partition logs; therefore, writing to disk becomes critical to achieve durability. By employing a more sophisticated logging scheme, it is straightforward to achieve better fault-tolerance guarantees. As this problem is orthogonal to our project goals, we are not planning to address this issue for the moment.

5.3 Materializer Layer

The materializer layer is a core component of Antidote. All high-level operations pass through this layer. Its responsibility is to generate snapshots of an object by applying operations and to reduce latency by avoiding access to the log on each read operation. To achieve these goals, each partition runs a materializer process with an in-memory cache, represented in the form of a key-value store mapping the key to a CRDT replica.

The materialization process and the caching mechanism are discussed in the following sections.

Materialization The materializer has to interpret the payload of the operations stored in the log and combine them to build CRDT states. This materialization process is done based on the meta-data encoding the consistency information in the read request parameter. For example, if the system wants to provide consistent snapshots of the object store, the materializer has to take care of generating an object view from the stored snapshots and operations it contains in a way that no operation which has been committed after the read's snapshot time is applied to it. Given a summary of the operations to include into a version, the materializer can compute the corresponding versions of the objects. To apply the updates for some object data type, the materializer is relying on some library providing functions to build CRDTs from operations and update existing object views.

Cache Other than handling object generation, the materializer operates as a write-through cache. It reduces read latency by avoiding accessing the log on each read operation. The cache is split into two parts:

- an *operation cache* where the most recent operations are stored, and
- a *snapshot cache* which stores the latest materialized views (snapshots) of a CRDT.

The materializer is involved in update and read operations. When the materializer receives an update operation, it first forwards it to the log layer to achieve persistence. After the write to the log has been acknowledged, the materializer forwards the acknowledgement to the client and then stores the operation to its operation cache. Note that there is no materialization process triggered at this stage.

When the materializer receives a read request, it checks whether there are suitable materialized views available in the snapshot cache and whether there are operations to be applied in the operation cache. If so, it materializes a new version of the CRDT that satisfies the request by applying the potentially missing operations to the corresponding snapshot, thus creating a new snapshot. In the event that a snapshot exists for operations in the cache, but none which satisfy the read request parameters, the materializer reads missing entries form the log and generates a corresponding snapshot.

Garbage Collection (GC) As new operations are stored and new snapshots of CRDTs are created and cached, the size of the in-memory cache grows. To keep the size bounded, it is necessary to implement mechanisms to clear both the operation and snapshot cache. This process is controlled by a separate, concurrently running process that is integrated into the materializer layer. The current implementation provides a very simple and general mechanism, but could be optimized for specific workloads.

As there are two cache components, interacting with any of them can generate the need for the garbage collection mechanism to be triggered.

- *Snapshot (or read-triggered) GC*: In the event that the number of snapshots stored for a key reaches the *snapshot_threshold*, the materializer keeps just the latest *snapshot_min* snapshots. Note that the snapshot generation is done on reads, hence this mechanism is triggered only by reads. All operations that are not included in the remaining snapshots and are more recent than the oldest one, remain in the operation cache while the rest are removed.

- *Operation (or write-triggered) GC:* In the event that the number of update operations stored for a key reaches the *ops_threshold*, the materializer triggers a read operation. The execution of this read operation will generate a snapshot and remove the operations from the cache that are included in the snapshot.

Failure Handling In the event of a failure, the materializer is restarted and restores its state by replaying the partition's log.

5.4 Transaction Layer

In addition to the traditional put-get interface of key-value stores, Antidote offers a transactional interface to clients. Transactions provide Transactional Causal+ Consistency semantics, as defined by the following properties:

- *Atomic writes*: The updates from a transaction are executed in an all-or-nothing manner.

- *Isolation*: Intermediate results of a transaction cannot be observed by other transactions; a transaction reads from a causally consistent snapshot.

- *Mergeable transactions*: Results from two concurrent transactions on the same key in different DCs can be merged.

An operation with a single update or a single read is internally also executed as a transaction.

Causal consistency with full replication Replication across multiple data centres (DCs) around the world is important for high availability and low latency access. Antidote supports replication of all objects fully across multiple data centres.

Transactional Causal+ Consistency[28, 50] allows transactions to execute in a DC without synchronising with other DCs. However, tracking causality across multiple partitions and multiple DCs is non-trivial. There have been different mechanisms proposed for causal consistency in the literature [28, 29, 17, 4, 18]. In short, many approaches suffer from unbounded size of meta-data needed to keep track of causality. Here, we discuss a protocol for Transactional Causal+ Consistency for Antidote which is a fully replicated partitioned database. Transaction management and replication protocol guarantees that the transactions are causally consistent, while the use of CRDTs permits the mergeability of transactions.

Related Work The ClockSI [18] protocol provides snapshot isolation (SI) for transactions in a partitioned data store without the need of a centralised entity to assign snapshot and commit timestamps. Instead, it uses loosely synchronised physical clocks to assign per-partition timestamps. Our protocol for transaction read and commit is inspired by ClockSI. We have adapted it to incorporate replication of datastore partitions across multiple DCs. Further, we rely on CRDTs for Transactional Causal+ Consistency instead of providing classical Snapshot Isolation semantics.

COPS [28] is a distributed key-value store that provides Causal+ Consistency. The causal dependencies are tracked per key resulting in large non-scalable meta-data. The dependency of an update is explicitly checked whether it is satisfied in the local data centre before writing it.

Orbe [17] is a protocol for causal consistency in partitioned and replicated data centres. The causality is tracked per update operation. The causality meta-data has size $O(P \times D)$, where P is the number of partitions and D is the number of data centres. The paper further describes mechanism to reduce the meta-data size such as dependency cleaning, which are employed similarly in other geo-replicated databases.

5.4.1 Causally Consistent Transactions

A client executes a transaction T consisting of an arbitrary number of reads and writes to multiple keys. Transactional Causal+ Consistency is achieved as follows: A transaction T executes on a consistent snapshot for all objects accessed. The snapshot of a transaction T denotes the causal dependencies of operations in T, and thus includes all updates covered by the snapshot.

Operations within a transaction T are of the following type:

- A *read* r returns all updates included in T's snapshot and all updates in T which precedes r. Reads are causally consistent.
- A *write* u is an update to a key. A write is not visible until committed.

Protocol Antidote's transaction protocol consists of two parts. The first part is handles the transaction management within a DC. The second one is responsible for eventually replicating the updates from a transaction to other DCs. Both of them together guarantee that the transactions are causally consistent.

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 timestamps for this partition and is loosely synchronized with clocks for the other partitions in the same DC.

- Each partition has a *partition vector clock* pvc_j^k with an entry corresponding to each DC. If $pvc_j^k[d] = t$, then partition p_j^k has received all updates committed on or before time t in partition p_d^k. $pvc_j^k[j] = t$ means that partition p_j^k has committed all transactions with commit time $\leq t$.

Meta-data per DC

- A *snapshot* s is represented by a vector having one entry for every DC. $s = S$ means that the snapshot contains all updates with $c \leq S$, from all DCs. The snapshot identified by a vector s is the same in all DCs. Every snapshot is causally consistent.
  A *stable snapshot* ss_j denotes the latest snapshot which is available in all partitions in DC j. For a stable snapshot, $ss_j[i] = t$ with $i \neq j$ implies that all partitions in DC j have seen at least all updates committed on or before time t from DC i.

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.

- Every transaction keeps a set $UpdatedPartitions$ of partitions that will be updated during the commit.

- Each client keeps a client vector clock cc with an entry corresponding to each DC. It keeps track of dependencies for client operations. All read and update operations should be executed on a snapshot with $cc \leq vs$.

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

- 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.

Algorithm 1 Transaction coordinator TC in partition k, DC j

```
function GetSnapshotTime(Clock $cc$)
    for all $i = 0..D-1, i \neq j$ do
        $vs[i] = ss_j[i]$
    end for
    $vs[j] = \max(pc_j^k, cc[j])$
    return $vs$
end function

function StartTransaction(Transaction $T$, Clock $cc$)
    for all $i = 0..D-1, i \neq j$ do
        wait until $cc[i] \leq ss_j[i]$
    end for
    $T.vs$ = GetSnapshotTime($cc$)
    return $T$
end function

function Update(Transaction $T$, Key $k$, Operation $u$)
    $p$ = partition($k$)
    $T$.UpdatedPartitions = $T$.UpdatedPartitions $\cup \{p\}$
    send ExecuteUpdate($T$, $k$, $u$) to $p$
end function

function Read(Transaction $T$, Key $k$)
    $p$ = partition($k$)
    send ReadKey($T$, $k$) to $p$
end function

function DistributedCommit($T$)
    for all $p \in T$.UpdatedPartitions do
        send Prepare($T$) to $p$
        wait until receiving ($T$, prepared, timestamp) from $p$
    end for
    CommitTime = max(received $timestamps$)
    $T.c = T.vs$
    $T.c[j]$ = CommitTime
    $T.dc = j$
    for all $p \in T$.UpdatedPartitions do
        send Commit($T$) to $p$
    end for
end function
```

Transaction Manager The transaction manager is responsible for executing transactions in a DC. Algorithm 1 shows the algorithm for a transaction coordinator running in DC j, which executes the transaction on behalf of clients. A client can contact any node in a DC and start a transaction coordinator TC. The

Algorithm 2 Transaction execution at partition m, DC j

```
1:  function ExecuteUpdate(Transaction $T$, Update $u$)
2:      wait until $T.vs[j] \leq pc_j^m[m]$
3:      log $u$
4:  end function
5:  function ReadKey(Transaction $T$, Key $K$)
6:      wait until $T.vs[j] \leq pvc_j^m[j]$
7:      return snapshot($K$, $T.vs$)
8:  end function
9:
10: function Prepare(Transaction $T$)
11:     prepareTime = $pc_j^m[m]$
12:     preparedTransactions$_j^m$.add($T$, prepareTime)
13:     send ($T$, prepared, prepareTime) to $T$'s coordinator
14: end function
15:
16: function Commit(transaction $T$)
17:     log ($T$, commit, $T.c, T.vs$)
18:     preparedTransactions$_j^m$.remove($T$)
19: end function
20:
21: function UpdateClock
22:     if preparedTransactions$_j^m \neq \emptyset$ then
23:         timestamps = Get prepare timestamps in preparedTransactions$_j^m$
24:         $pvc_j^m[j] = min$(timestamps) $- 1$
25:     else
26:         $pvc_j^m[j] = pc_j^m$
27:     end if
28: end function
```

client then issues update and read operations via TC.

When transaction T starts, it is assigned a snapshot time vs. It is desirable to assign the latest snapshot available in the local DC as vs, so that the transaction reads the latest updates. However, as we will see in the replication protocol, each partition is replicated independently of other partitions. This may result in some partitions having more recent snapshots and other partitions having older snapshots. Hence the ss_j which is available in all partitions is assigned to the vs of the transaction. Since the clocks of partitions within the same DC are less likely to be out of sync for a long time, we can assign $vs[j]$ to be the physical clock of the partition running the transaction coordinator, so that T can see the latest committed transactions in DC j.

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 waits until the stable snapshot of DC j has reached cc. Thus, using a client clock cc guarantees that a client always observes monotonic snapshots even when connecting to other DCs.

After transaction snapshot time is assigned, the client can issue operations. These operations will be forwarded to the partitions of the key involved in the operation. Using the stable snapshot vector ss_j to assign $T.vs$ guarantees that all partitions have received all updates from other DCs required by T's snapshot. Hence the read operation has to check only if the updates from DC j required for the snapshot are available in the partition. The read protocol waits for $pvc_j^k[j]$ to become T's snapshot time to commit, to ensure that updates from the transaction's in prepared phase which must be in T's snapshot are also included in the read. The update protocol waits until the physical clock of the partition dominates T's snapshot time and log the update.

During the commit phase, the coordinator requests prepare timestamps from partitions involved in the transaction. The maximum of the prepared timestamps is assigned as the commit-time by the coordinator. The commit-time defines a new snapshot of the DC. Hence $T.c$ is derived from the scalar commit time decided by the coordinator and $T.vs$.

Replication Protocol The replication protocol is responsible for replicating updates committed in one DC, to other DCs. In the protocol, each partition p_j^m sends updates to p_i^m, for $i = 0$ to $d-1$ DCs, $i \neq j$. The main idea behind the protocol is to keep each partition causally consistent. This is guaranteed by following conditions.

- A transaction T from p_i^m is applied in p_j^m, only if T's causal dependencies are satisfied locally in p_j^m.

The next condition is that each partition has an increasing clock.

- Each partition p_i^m in DC i sends transactions to p_j^m in DC j in commit time order. If p_j^m receives a transaction T from p_i^m, where $T.c = (i, c)$, p_j^m must have received all transaction with commit time (i, v), such that $v[i] < c[i]$ from p_i^m. When the transaction is applied, the partition vector clock $pvc_j^m[i]$ is set to $c[i]$.

Given the above two conditions are satisfied, if $pvc_j^m = V$, any snapshot $S \leq V$ is available in partition p_j^m.

If a partition p_j^m does not execute any new transaction for a long time, the remote partition p_i^m's entry $pvc_i^m[j]$ will not be increased, resulting in a large gap compared to vector clocks of other partitions. In order to avoid this, partitions send a periodic heartbeat message with their vector clock to other remote partitions.

The algorithm for sending updates from DC i to j is given in Algorithm 3. Algorithm 4 shows the process of handling updates from DC i received at DC j.

Given that each partition is causally consistent and pvc_j^m denotes all snapshots available in partition p_j^m, the transaction manager ensures that a transaction executes on a causally consistent snapshot always.

5.5 Implementation

In this section, we briefly introduce concepts and technologies that we use to develop and test Antidote.

Algorithm 3 Replication Algorithm at the sender, running in partition p_i^m

```
function ReplicateToDC($j$)
    loop
        $t = pvc_i^m[i]$
        Transactions = $\{T \mid T.c[i] \leq t, T.dc = i,$ not propagated to DC $j$ yet $\}$
        if Transactions $= \emptyset$ then
            heartbeat = new Transaction()
            heartbeat.$c[i] = t$
            heartbeat.$dc = i$
            heartbeat.$vs = pvc_i^m$
            send heartbeat to $p_j^m$
        else
            sort Transactions in ascending order of $T.c[i]$
            send Transactions to DC $j$
        end if
    end loop
end function
```

Algorithm 4 Replication Algorithm at the receiver, running in partition p_j^m

```
queue[i]: A queue for transactions received from DC i

function ReceiveTransaction(ListofTransactions Transactions, DC $i$)
    for all $T$ in Transactions do
        enqueue(queue[$i$], $T$)
    end for
end function

function ProcessQueue(i)
            ▷ This function is repeatedly called to process transactions from DC $i$
    $T$ = getFirst(queue[$i$])
    $T.vs[i] = 0$
    if $T.vs \leq pvc_j^m$ then
        if $T$ is not a heartbeat then
            log $T$
        end if
        $pvc_j^m[i] = T.c[i])$
        remove $T$ from queue[$i$]
    end if
end function

function calculateStableSnapshot
    for all $j = 0..D-1$ do
        $ss_i[j] = \min_{k=0...P-1} pvc_i^k[j]$
    end for
end function
```

Erlang OTP Erlang OTP provides a set of libraries for developing scalable distributed systems. Antidote is written using the Erlang programming language and respecting the OTP design principles that define how to structure Erlang code in terms of processes, modules and directories
(cf. `http://www.erlang.org/doc/design_principles/des_princ.html`).

Riak libraries and components riak_core is a toolkit for building distributed, scalable, fault-tolerant applications. It is an Erlang OTP application that provides a number of services useful for writing distributed applications, namely node liveness and membership, partitioning and distributing work, and managing cluster state. Antidote is build on top of riak_core. Using riak_core, Antidote partitions the set of keys onto different nodes. The functionalities per partitions such as log, transaction execution etc. are implemented as a virtual node (vnode) provided by riak_core.

riak_dt provides a set of state-based CRDTs implemented in Erlang. Protocol buffers (`https://github.com/google/protobuf/`) is a library for standardized message serialization. Antidote exposes a protocol buffer interface besides RPC, through which clients can access the distributed data store.

Testing and benchmarking facilities riak_test is a system for testing riak_core clusters. Tests are written in Erlang, and can interact with the cluster using distributed Erlang. Basho Bench is a benchmarking tool for performing accurate and repeatable performance tests and stress tests, and produce performance graphs. It focuses on two metrics of performance: throughput and latency.

5.5.1 Setting up Antidote

In this section we explain how to build and test Antidote.

Prerequisites

- A unix-based OS
- Erlang R16B02

Building Antidote

1. Clone Antidote from the github repository.

   git clone http://github.com/SyncFree/antidote

2. Go to the antidote directory (the one that you've just cloned using git) and build the system using *make rel*. This will now pull all the dependencies it needs from github, build the application, and finally make an erlang "release" of a single node. If all went well, you should be able to start a node of Antidote.

   rel/antidote/bin/antidote start

Using Antidote We can interact with Antidote directly using Distributed Erlang and remote procedure calls.

- To start a client node

  erl -name 'client@127.0.0.1' -setcookie antidote

- First check whether client can connect to the cluster:

  net_adm:ping('antidote@127.0.0.1').
  pong

- Perform a write operation (example):

  rpc:call('antidote@127.0.0.1', antidote, append,
  [myKey, riak_dt_gcounter, {increment, actor}]).
  ok,1,'dev1@127.0.0.1'

  where *myKey* is the key to write to and*riak_dt_gcounter* is the type of the Key.

- Perform a read operation (example):

  rpc:call('antidote@127.0.0.1', antidote, read, [myKey, riak_dt_gcounter]).

Antidote can also be accessed using its protocol buffer interface.

- Start an erlang console with the required dependencies:

  erl -pa antidote/deps//ebin/ antidote/ebin/*

- Connect to the database

  {ok, Pid} = antidotec_pb_socket:start("localhost", 8087).

- Read or create a new key with a counter data-type:

  Obj = antidotec_pb_socket:get_crdt(Key, riak_dt_pncounter, Pid).

- Increment and read the value of the counter:

  Obj2 = antidotec_counter:increment(Obj).
  antidotec_counter:dirty_value(Obj2).

- Store the updated object:

  antidotec_pb_socket:store_crdt(Obj2, Pid).

More information, tests, and benchmarks can be found at

`http://github.com/SyncFree/antidote`

5.6 Summary and Outlook

In this Section, we introduced Antidote, the common research platform for the SyncFree project. The first version of the platform provides a CRDT datastore running on a small and stable number of data centres. Clients can interact with the datastore by submitting read requests and update operations, possibly grouped together using transactions with Transactional Causal+ Consistency semantics.

During the first months of the project, we gave particular attention to the architectural design of Antidote to be compliant with future adaptations and integrations of results from the other WPs.

Special attention was given to chose, adapt and implement the inter-DC replication mechanisms. We plan to soon submit a paper on the involved protocols, thus presenting the platform to the research community.

Applying the testing and benchmark tools, we are currently evaluating possible bottlenecks in the implementation of Antidote. In particular, the extensive tests already led to several improvements and bug fixes.

More on planned improvements can be found in Section 8.

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

Client-side applications, such as in-browser and mobile apps, are not well served by current technology for wide-area data sharing. Existing systems either do not offer sufficient consistency and availability guarantees, or do not scale to large numbers of client devices, or both. For instance, access to data stored in the cloud incurs a round-trip to the data centre, which is subject to wide-area latency [19], unavailability, and lack of session guarantees [42]. This results in headaches for app developers, who resort to implementing their own ad-hoc replication layer.

For the second milestone, WP2 targets a setting where the distributed datastore takes the responsibility of ensuring correct, scalable access to client-side applications by managing a partial replica close to clients at dedicated points of interest. In particular, it should ensure consistency, availability, and convergence at least as well as recent geo-replication systems [28, 29, 17]. Under the availability requirement, critical to many client apps, the strongest consistency model is *Causal+ Consistency* [32].

Supporting thousands or millions of client-side replicas challenges the assumptions of existing Causal+ algorithms.

- Algorithm that track causality per client replica [7, 33] lead to meta-data whose size grows unacceptably, but offloading causality-tracking to servers has fault-tolerance issues.
- Scalability to high numbers of small devices requires to replicate data and metadata only partially at those devices [7]; but under partial replication, clients replicas may suffer gaps in causality.
- Algorithms that assume that application is located inside a data centre (DC) [28, 29, 17], for instance to ensure session guarantees, are inapplicable.
- Metadata compaction algorithms that require stability [33, 28, 29, 17], become unavailable with many failure-prone replicas.

In anticipation of Task 2.2 in SyncFree, we continued and extended our work on SwiftCloud, a system that addresses the challenges of client-side replication. 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. Our design demonstrates how to make use of the presence of a client replica, and the fact that the local application session is tied to it, into an advantage other than low latency. Our insight is to *write fast* into the local replica, and when convenient *read in the past* slightly stale data. This allows SwiftCloud to have small meta-data, to improve both latency and throughput, and to remain available, without affecting consistency.

Protocols with decoupled, bounded meta-data. 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 funnelling updates through DCs, the size of meta-data remains small and stable, at the expense of staleness, but without affecting correctness. This work is building on the ideas of Dotted Version Vectors, further described in the WP3 report.

Partial-replica fail-over protocol. When a client connects to different DCs (e.g., because of failure), they may be mutually inconsistent. SwiftCloud let 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.

Always-available log pruning. Logs must be eventually pruned. In prior systems [33, 7], pruning was unsafe if a replica was unavailable, or caused large vectors to be transmitted. In our design, any prefix of the log known by all DCs can be replaced with a shared meta-data protecting from duplicates.

Scale-out DCs with gapless vectors. To allow parallelism inside the DC, without causing gaps in timestamp vectors [34], SwiftCloud exposes only states for which there are no gaps.

Our evaluation of SwiftCloud on EC2 and PlanetLab shows that the system provides immediate and consistent client-side access for replicated (cached) objects, with small and bounded metadata, and that the cost in staleness is low.

6.1 Fault-tolerant session and durability

We discuss now how SwiftCloud handles network, DC and client faults, focusing on client-side mergeable transactions. When a client loses communication with its current DC, due to network or DC failure, a client may need to switch over to a different DC. The latter's state is likely to be different, and it might have not processed some transactions observed or indirectly observed (via transitive causality) by the client . In this case, ensuring that the clients' execution satisfies the consistency model and the system remains live is more complex. As we will see, this also creates problems with durability and exactly-once execution.

Causal dependency issue 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 [12]. Others remain asynchronous or rely on a single DC, but after failover clients are either blocked or they violate causal consistency [28, 29, 27]. 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. Since, as explained earlier, a client cannot keep track of all transitive dependencies, we restrict the set of dependencies. We define a transaction to be K*-durable* [33] at a DC, if it is known to be durable in at least K DCs, where K is a configurable threshold. Our protocols let a client observe only the union of:

- its own updates, in order to ensure the "read-your-writes" session guarantee [42], 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.

Durability and exactly-once execution issue A client sends each transaction to its DC to be globally-committed. The DC assigns a DC timestamp to the transaction, and eventually transmits it to every replica. If the client does not receive an acknowledgment, it must retry the global-commit, either with the same or with a different DC. However, the outcome of the initial global-commit remains unknown. If it happens that the global commit succeeded with the first DC, and the second DC assigns a second DC timestamp, the danger is that the transaction's effects could be applied twice under the two identities.

For some data types, this is not a problem, because their updates are idempotent, for instance `put(key,value)` in a last-writer-wins map. For other mergeable data types, however, this is not true: think of executing `increment(10)` on a counter. Systems restricted to idempotent updates can be much simpler [29], but in order to support general mergeable objects with rich merge semantics, SwiftCloud must ensure exactly-once execution.

Our approach separates the concerns of tracking causality and of uniqueness, following by the insight of [3]. Recall that a transaction has both a client timestamp and a DC timestamp. The client timestamp identifies a transaction uniquely, whereas the DC timestamp is used when a summary of a set of transactions is needed. Whenever a client globally-commits a transaction at a DC, and the DC does not have a record of this transaction already, the DC assigns it a new DC timestamp. This approach makes the system available, but may assign several DC timestamp aliases for the same transaction. All alias DC timestamps are equivalent in the sense that, if updates of T' depend on T, then the dependencies of T' comprise the dependencies of T, involving at least one of the aliased ids. The system then guarantees that T' will execute after T in every replica.

When a DC processes a commit record for an already-known transaction with a different DC timestamp, it adds the alias DC timestamp to its commit record on durable storage.

To provide a reliable test whether a transaction is already known, each DC maintains durably a map of the last client timestamp received from each client . Thanks to causal consistency, this value is monotonically non-decreasing. Thus, a DC knows that a transaction being received for global-commit from a client has already been processed if the recorded value for that client is greater or equal to the client timestamp of the received transaction.

6.2 Implementation and experimental evaluation

Our experimental study on a prototype shows that our design reaches its objective at the modest cost in staleness. We evaluate SwiftCloud, on EC2 and PlanetLab, against the social network app of [41] and against the YCSB benchmark [11]. When data is cached, both reads and updates return immediately. Response time, for updates to data missing in the cache, is two orders of magnitude lower than for synchronous protocols with similar availability guarantees. With 3 DCs, the overhead of meta-data is almost constant, at the level of max. 40 bytes per update delivered to and from the client. When a DC fails, its clients switch to a new DC in under 1000 ms. The throughput for updates scales with the number of servers in a DC (up to 5 servers, without impacting meta-data size. In benign runs, under $K = 2$, which covers most common failure scenarios, aggregated staleness overhead stays fewer than 1% of operations read stale data.
More details on SwiftCloud, including an extensive evaluation can be found in the paper in Appendix B.

7 Adaptive replication

The amount of data being processed in data centres (DCs) keeps growing at enormous rate [44, 10, 9]. Some of the areas where the amount of stored data already reach terabytes (TBs) and even petabytes (PBs) are data mining, social networks[1], particle physics, climate modeling, high energy physics and astrophysics, all data which needs to be shared and analysed [25, 35, 36]. Yet, the location of a DC relative to the client accessing the data has an impact on availability, access times (latency - accessibility) and costs. Replicating data at multiple sites provides a solution to undesirable effects [8, 1, 45]: Replication may result in reduction of bandwidth usage and decrease of user response time on data access. But keeping (too) many replicas of the data incurs extra costs, such as additional network traffic for keeping all replicas consistent under concurrent updates, additional storage, and additional computational power [21].

Replication strategies can be classified into two main groups:

- *static replication* where a replica persist until it is deleted by a user or its lifetime expires, and
- *dynamic replication* where the creation and deletion of a replica are managed automatically by the system and is normally directed by the access pattern of the data used by the users [15].

Targeting the second milestone for the SyncFree project, we want to find an replication strategy that minimises infrastructure utilisation under different access patterns.*Adaptive geo–replication* is a form of dynamic replication that deals with the problem of where the data (or parts of the data) is located within the network of DCs and how many replicas exist simultaneously [23, 25, 48, 1, 2, 30].

We propose an adaptive algorithm which is based on Wolfson's algorithm, [49], which is an adaptive algorithm for replicated data between processors taking into account changes in the read-write pattern. It further follows the principles of the Ant Colony Optimisation algorithms, which are inspired by the behavior of ant colonies when deciding which path to follow when foraging [16].

Note that the purpose of adaptive geo-replication is not disaster recovery. This safety mechanism should be provided by an orthogonal scheme focusing on other characteristics such as the geographical distribution of DCs.

7.1 Algorithm

The general idea of this algorithm is to decide without the need of human intervention where and when to replicate some data item while keeping into account the benefits and costs. To provide availability, every data item must be present (replicated) in at least in one of the DCs. Any read operation performed in a DC reinforces the need for a replica of the data in this DC. Similarly, this holds for update operations. However, updates make it less desirable to have replicas of this data in other DCs due to the overhead of propagating the updates.

[1]Facebook generates up to 4 PB of data per day that must be modifiable, see `https://research.facebook.com/blog/1522692927972019/facebook-s-top-open-data-problems/`

The algorithm is explained in more detail below. The variables and constants used in the explanation are summarised in Figure 2.

Let $X_{dk} = 1$ denote the existence of a replica of data item k in DC d, while its absence in this DC is given by $X_{dk} = 0$. The total number of replicas for data item k is then given by

$$R_k = \sum_{d=1}^{|DC|} X_{dk}$$

Let F_{kd} denote the actual replication strength of data item k in DC d. It is defined as

$$F_{kd} = \max(0, \min(L_k, r_{kd} * \Delta r_k + w_{kd} * \Delta w_k - \sum_{i=1, i \neq d}^{|DC|} X_{kd} * w_{kd} * \Delta w_{kdi} - X_{kd} * \Gamma))$$

The equation reflects that the replication strength is increased by the reads and writes requested through DC d, with intensities Δr_k and Δw_k, respectively. It is weakened by the writes requested through other DCs with intensity Δw_{kdi}. Furthermore, it is weakened by a temporal decay factor in last term of the equation. The model implies that only DCs with a replica of the data will be penalised by writes and the time factor Γ.

Figure 3 illustrates the relation of the replication strength F_{kd} and the thresholds for keeping (T_k^+)and removing (T_k^-) a replica at some DC d.

A replica is created at a DC d once the replication strength F_{kd} reaches the threshold T_k^+. If the replication strength falls below the threshold T_k^-, the replica is removed at the DC only if the total number of replicas for this data item would then still be higher than the required minimum N_k. Therefore, every DC containing a replica for some data item must know about the other DCs that keep replicas of this data item.

The factor Δw_{kdi} represents the decay of replication strength in DC i as consequence of a write request in another DC d. For a simple model of this factor, we assume that Δw_{kdi} increases with the network distance between DCs d and i. Further, we assume that the value is symmetrical, i.e. $\Delta w_{kdi} = \Delta w_{kid}$, so that it incurs the same cost to transfer the data from DC d to DC i as to transfer from DC i to DC d.

The effect of the time-based factor Γ is required to ensure that, in absence of updates, the replication factor for a data item still decreases over time. In particular, if the reads are concentrated in a few DCs, then the replicas in other DCs are eventually removed, retaining at least N_k replicas.

A read request to a DC, which does not have a replica of the data, is forwarded to the closest DC with a replica. This second DC does not gain strength from the read operation as the read was not initiated originally at this DC. Until the threshold T_k^+ has been reached, the first DC acts as a proxy for the data access. Once the threshold is reached, the DC notifies all other DCs holding a replica about the existence of the its own replica.

Variable	**Description**	**Type**
DC	Set of all DCs. d identifies one of the DCs, $d \in \{1, \ldots, \|DC\|\}$. DC$_d$ represents the DC identified by d.	$d \in \mathbb{N}^+$
D_{kd}	Replica of data item k in DC d, $k \in \{1, \ldots, \|D\|\}$.	$k \in \mathbb{N}^+$
r_{kd}	Number of reads for data item k requested at DC d	$r_{kd} \in \mathbb{N}_0$
Δr_k	Increase of replication strength in a DC when executing a read	$\Delta r_k \in \mathbb{R}^+$
w_{kd}	Number of writes for data item k requested at DC d	$w_{kd} \in \mathbb{N}_0$
Δw_k	Increase of replication strength in a DC when executing a write	$\Delta w_k \in \mathbb{R}^+$
Δw_{kdi}	Decay of replication strength in DC i as consequence of a write request in DC d. This value depends on both DCs d and i and may also depend on other factors (e.g. time of the day).	$\Delta w_{kdi} \in \mathbb{R}^+$
Γ	Decay of replication strength with time. Example: Linear decay with factor τ since the creation is given by $\Gamma = \Delta t * \tau$	$\Gamma \in \mathbb{R}^+$
N_k	Minimum number of replicas of data k with $1 \leq N_k \leq \|DC\|$, default $N_k = 1$	$N_k \in \mathbb{N}^+$
T_k^+	Threshold for replication strength required to start the replication of data k in a DC currently not containing a replica	$T_k^- \in \mathbb{R}^+$
T_k^-	Threshold for replication strength below which a replica of data item k is removed in a DC currently containing a replica, default $T_k^- = 0$	$T_k^- \in \mathbb{R}^+$
L_k	Maximum replication strength for data item k	$L_k \in \mathbb{R}^+$
F_{kd}	Replication strength for data item k in DC d, $F_k \leq L_k$	$F_k \in \mathbb{R}^0$
R_k	Number of replicas for data item k, with $1 \leq N_k \leq R_k \leq \|DC\|$	$R_k \in \mathbb{N}^+$
X_{kd}	Existence of a replica of data k in DC d, with value 1 if the replica exists, and 0 otherwise	$X_{kd} \in (0, 1)$

Figure 2: List of variables and constants.

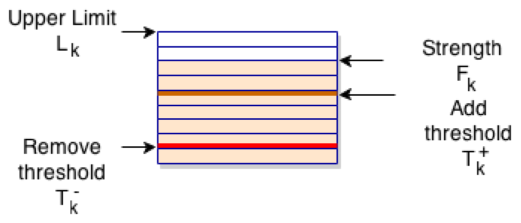


Figure 3: Relation of thresholds and replication strength at a DC.

Extending this basic replication model, it would be possible to add another temporal effect, the Time To Live (TTL), to make sure that eventually the data item will be fully removed. 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 normal DCs.

The algorithm is optimal in the sense that when the replication scheme stabilises, the total number of replicas required for the reads and writes is minimal.

Figures 4 and 5 present sequence diagrams and a flowchart for the read protocol. Similarly, Figures 6 and 7 details the write protocol. Note that the second figure in Figure 4 does not show the notification step when a new replica is created.

The initial creation of a data item generates a replica in the directly accessed DC, as the number of replicas must be at least one ($N_k \geq 1$). If $N_k > 1$, then the system should generate and distribute additional replicas in the close-by DCs. This part of the replication protocol could be integrated with other initialization processes such as checking for the existence of the proposed key when the system provides unique keys.

Discussion For typical data access pattern,the number of reads is higher than the number of writes. Further, we assume that a read should be executed before a write. For this scenario, Δw_k could be chosen to be smaller than Δr_k ($\Delta w_k < \Delta r_k$) so that more writes are required to maintain or create a replica in a DC.

Some of the parameters may be generalised even further by allowing them to depend on the DC where some calculation is executed, i.e. using some Δr_{kd} instead of Δr_k, and similarly Δw_{kd}. To determine the replication strength, other factors like bandwidth or storage capacity could be added. Paiva et al. [37] use a very simple representation of the available storage capacity. For now, we assumed that the capacity in a DC is sufficient high, and when more storage capacity is required, extra storage can be added to the DC.

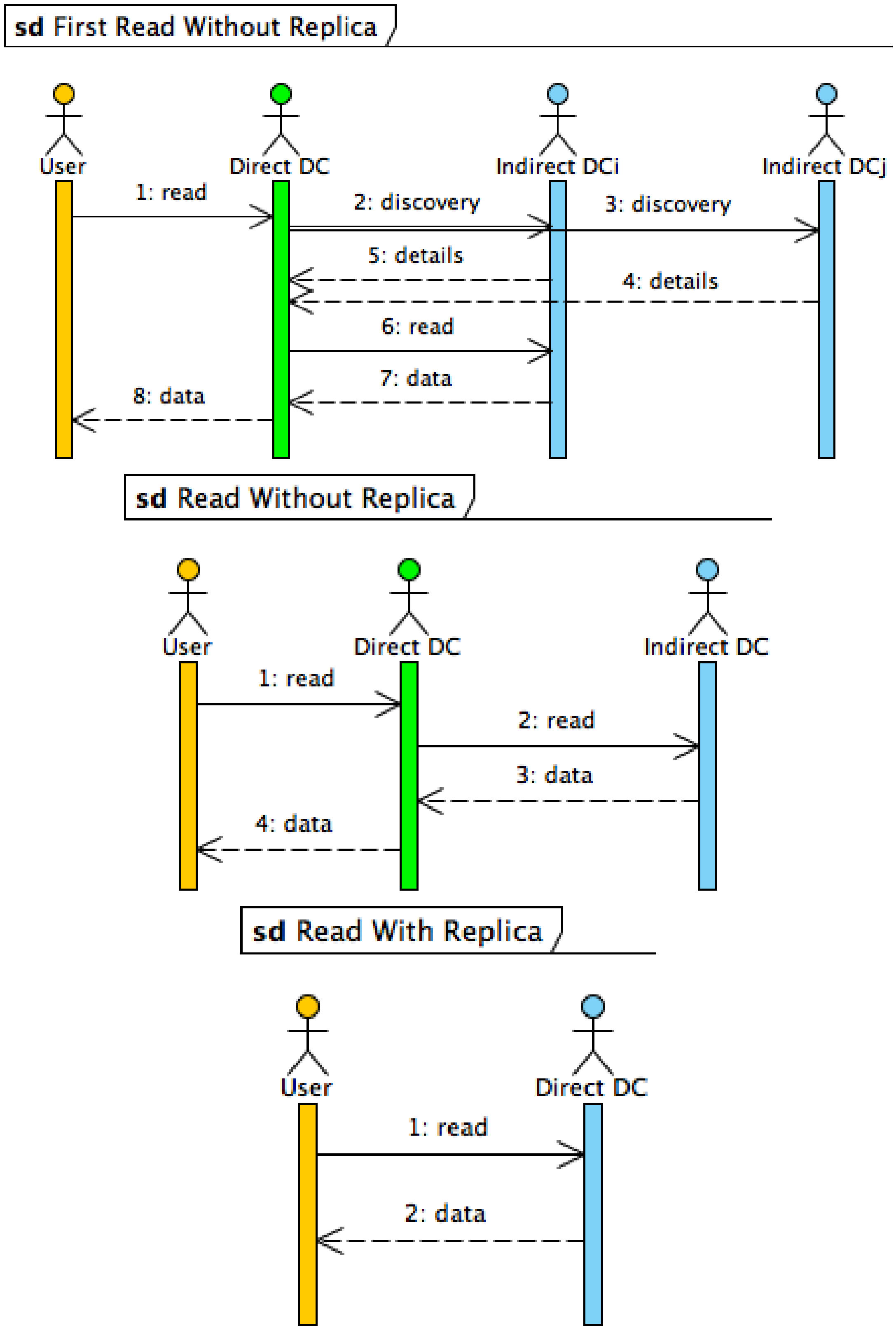


Figure 4: Read Sequence Diagrams.

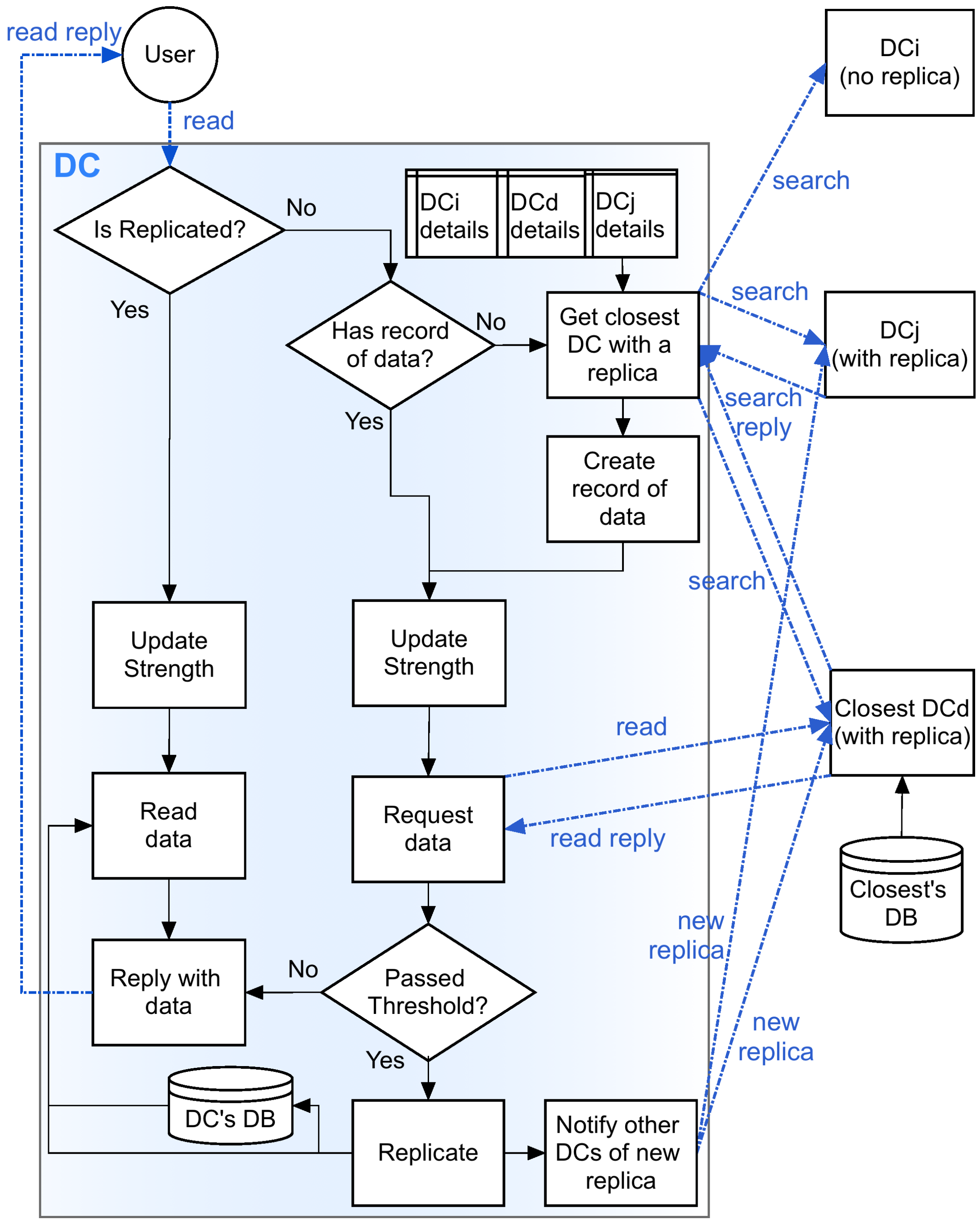


Figure 5: Flowchart for read requests.

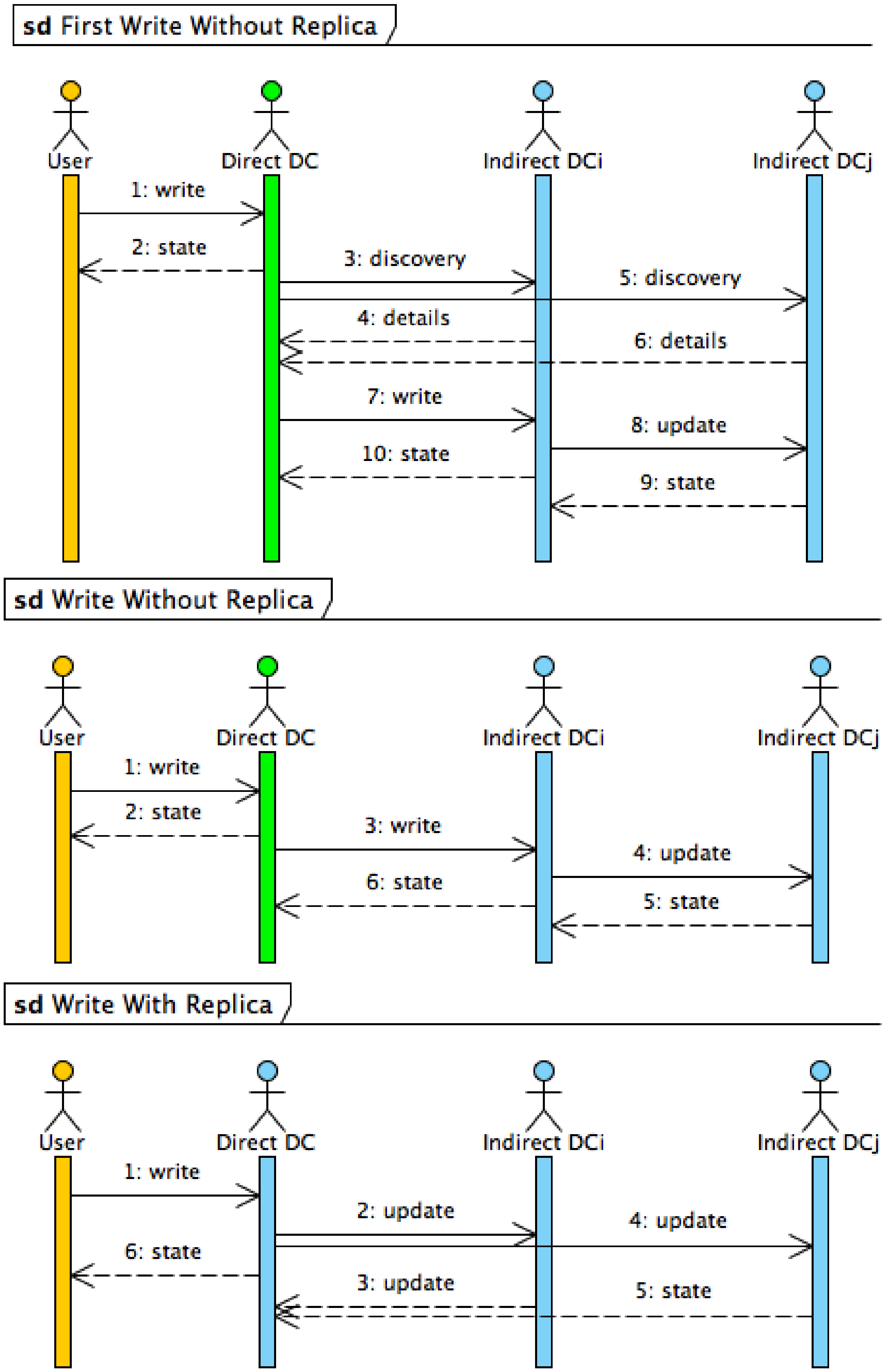


Figure 6: Write Sequence Diagrams.

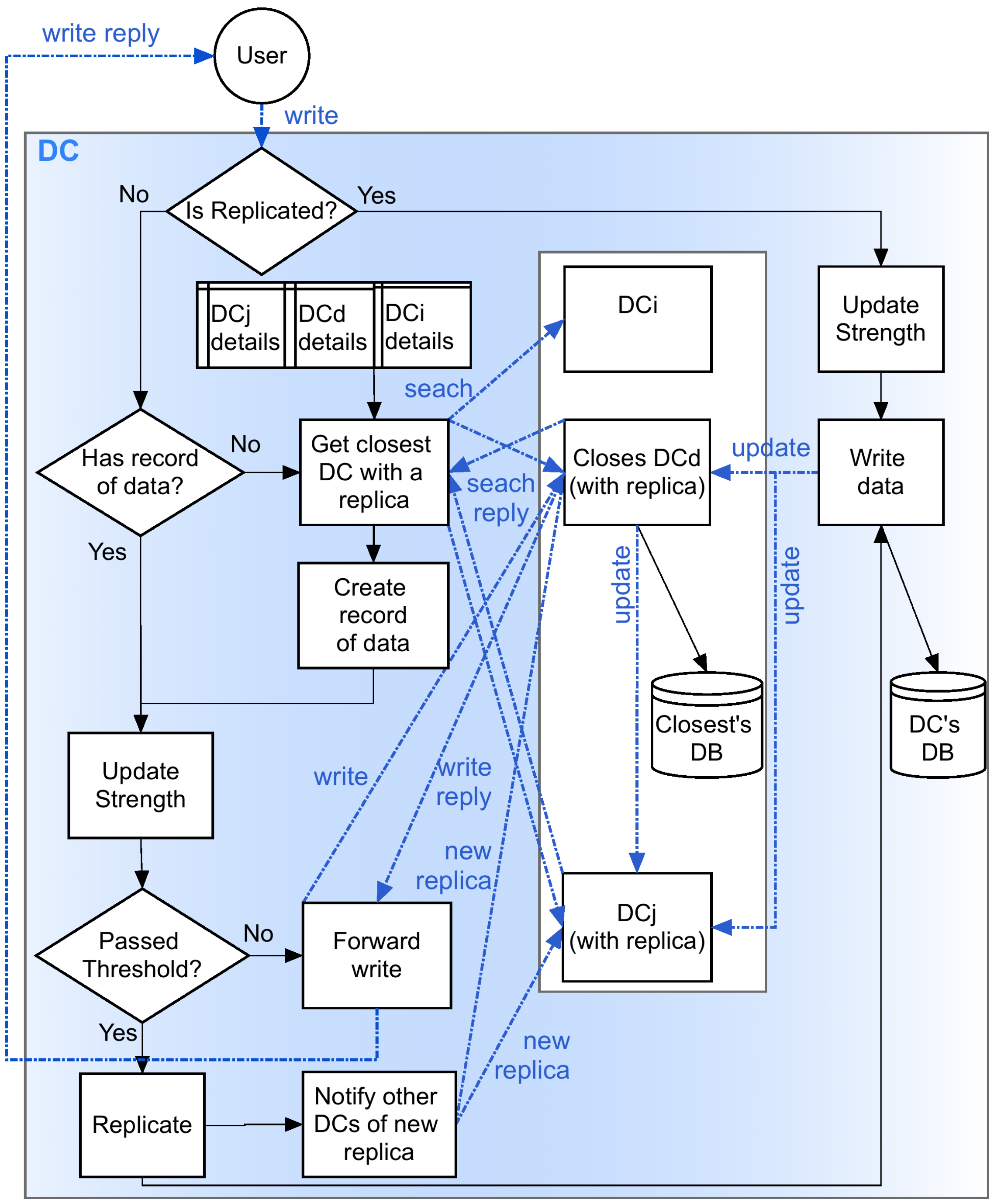


Figure 7: Flowchart for write requests.

7.2 Comparison of replication schemes

We now want to compare three different replication strategies for the different accesses. To this end, the following schemes are analysed:

A. Only one fixed replica, i.e. no replication as such.

B. Adaptive location of replicas as described in Section 7.1.

C. Full replication, i.e. one replica in every DC.

Table 1 shows the steps involved for the operations in the sequence diagram for the respective scheme. Regarding the storage, using more replicates increases the storage requirements. For reads, the existence of the data in several DCs reduces the total number of operations required as it is more likely that the initially accessed DC already has a replica of the data compared to when the data only exists in one DC. For writes, having the data replicated in multiple DCs increases the total number of operations proportionally to the number of replicas. One technique to reduce the number of writes would be to combine multiple write in one before forwarding it to other DCs.

All these schemes can actually be achieved by adapting the minimum number of replicas and threshold values in the proposed algorithm:

- **One fixed replica**: $N_k = 1, T_k^+ = \infty, T_k^- = -1$.
- **Full replication**: $N_k = |DC|$.

This means that systems where the number of reads is significantly higher than the number of writes will benefit from replicating the data in multiple DCs, whereas systems with similar of higher number of writes will benefit from low or no replication (only one replica). However, this is a rather simplified view as there are more requirements that have an important influence on the selection of the actual system configuration. Some of the most common requirements are Scalability, Accessibility, Latency and Security.

- **Scalability**: Scheme (A) does not provide scalability in contrast to the actual replication schemes. Jimenez-Peris et al. [24] provide an analytical study which shows the scalability limits of full replication as updates have to be sent and executed at all replicated sites (symmetric processing). To reduce this cost, one could use asymmetric processing where transactions are processed first at the originating site, then collected, and eventually propagated and applied to the other sites, thus improving scalability. From the point of view of scalability, the processing power in a DC, when a write is received, is invested in processing the write and the actual updates. As the number of replicas increases, there is a point at which the increase on the number of replicas does not increase any more the total system capacity. The main reason is that most of the system processing power is then used in processing the updates.

- **Availability/Accessibility**: Schemes (B) and (C) improve accessibility, while scheme (A) provides only limited accessibility, see [26].

Sequence	**One fixed replica (A)**	**Algorithm (B)**	**Full replica (C)**
Storage	$1*$ data	$R_k *$ (data + info) + Δinfo	$\|DC\|*$ data
First read without replica	$2*$ reads $+(\|DC\|-1)*$ discoveries	$2*$ reads $+(\|DC\|-1)*$ discoveries	$1*$ reads
Read without replica	$2*$ reads $+(\|DC\|-1)*$ discoveries	$2*$ reads	
Read with replica	$1*$ reads	$1*$ reads	
First write without replica	$2*$ writes $+(N_k-1)*$ discoveries	$(R_k+1)*$ writes $+(N_k-1)*$ discoveries	$\|DC\|*$ writes
Write without replica	$2*$ writes $+(N_k-1)*$ discoveries	$(R_k+1)*$ writes	
Write with replica	$1*$ write	R_k* writes	
Create data	$1*$write	N_k* writes	$\|DC\|*$ writes

Table 1: Required operations by the different implementations for the specified sequences, $1 \leq N_k \leq R_k \leq |DC|$.

- **Latency**: As schemes (B) and (C) provide replicas in multiple DCs, the data can be accessed from any of those DCs. Choosing the one which is closest improves the latency experienced by the customer, thus improving responsiveness.
- **Security/Fault-tolerance**: Security is usually achieved by setting up recovery data centres, but it is also improved/achievable by the provision of replicas by means of redundancy [22, 47].

7.3 Summary

In this Section, we introduced an algorithm for adaptive geo–replication where the replication scheme is dynamically modified based on the current replication strength for some data item.

1. For reads, if the DC to which the clients connects has a replica of the requested data item, the DC does not need to communicate any information to any of the other DCs.

   In case a read is executed on a DC without a replica of the data, storage and processing power in the DC will be required and the request is forwarded to its closest neighbor DC with a replica. This operation incurs additional network traffic, but also increases the replication strength of the data, thus making it more likely that the data item is eventually replicated in the DC contacted first.
2. For writes, the DC receiving the original request will (eventually) transmit it to the other DCs, which have a replica of the data. Depending of the type of CRDT used, i.e. op-based or state-based, different methods for update propagation could be employed.
3. If data is not accessed frequently, the number of replicas is reduced as time passes, due to the temporal effect (Γ), until the data is only replicated in a minimal number of DCs.

To adapt replication schemes to access patterns requires simple and fast replica generation and removal. Furthermore, values for the parameters for calculating the cost could be determined and continuously adpated at runtime by some learning algorithm.

The work on adaptive replication spans several work packages: WP1 helped to infer the requirements and desirable characteristics for replication schemes; in WP2 we are developing protocols to implement adaptive replication and integrate it into the Antidote platform; WP5 contributes a visualisation and testing tool to evaluate different replication strategies. In the next section, we give an outlook on future work regarding adaptive replication, in particular the integration with Antidote.

8 Outlook for WP2

For the next months, we plan to progress in WP2 in several directions.

Improving and extending Antidote The Antidote platform as of now contains a prototype implementation for every component as presented in Section 5. Applying the recent research results from within the project consortium, we are continuously adapting and modifying Antidote. For the next 6 months, we plan to improve in the following way:

- We are currently investigating how to apply the techniques for pure op-based CRDTs to Antidote's materializer cache to overcome some performance issues we detected. The contributions of this work will be a new optimized op-based CRDT library[2] , an improved cache layout based on partially ordered logs and corresponding garbage collection mechanisms.

- WP3 proposes mechanisms for enforcing invariants on CRDTs while restraining from strong synchronization. This work complements Antidote's transactions which provide weak cross-CRDT guarantees. To consolidate these results, we will integrate the reservation/escrow technique into Antidote in the next couple of months to fully address the WP1 use cases "Wallet application" and "Ad counter".

- Similarly, we are about to implement composition mechanisms developed in WP3 and WP4, such as a Map CRDT and the techniques for partially replicating large CRDTs.

CRDTs under partial replication In anticipation of the next milestone, we already started in parallel to work on tracking causal consistency in partially-replicated datastores. In comparison to the layout of the current Antidote architecture, such a datastore is not fully replicated at every DC. Tracing the causal dependencies is considerably more difficult in this setting because the transitive closure of the causality relation needs to be determined even though elements in this relation might not be available locally.

The work on adaptive replication (see Section 7) layed the foundation for this work by showing how replication schemes can be adapted to different, dynamically changing access patterns to the objects. However, we need to investigate further the principles and mechanisms of partitioned (or shareded) databases to provide fast and correct implementations of partial replication schemes.

The algorithm for causal consistency which we are working on is based on using vector clocks. Each vector clock contains an entry for every DC subsuming all shards within this DC. As a next steps, we will refine the protocols to reducing the size of the vector clock in order to decrease the meta-data overhead.

Further, the protocols for consistent partial and adaptive replication will be integrated in Antidote, thus providing different platforms for different system setups.

[2]As there is no op-based CRDT library in Erlang available yet, we are currently using the state-based riak_dt library to simulate op-based CRDTs.

All platforms will share the CRDT code base, the logging system and materializer, and parts of the code for transactions, but provide new or extended implementations for components dealing with replication strategies and update propagation.

9 Papers and publications

The following papers have emerged from the work in WP2:

- Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In *Distributed Applications and Interoperable Systems (DAIS)*, pages 126–140. Springer, 2014 (Appendix A)
- Marek Zawirski, Nuno Preguiça, Sergio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. Under submission. (Appendix B)
- Nuno Preguiça, Marek Zawirski, Bieniusa Annette, Paulo Sérgio Almeida, Valter Balegas, Carlos Baquero, and Marc Shapiro. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *Proceedings of the Workshop on Planetary-Scale Distributed Systems*. Springer, 2014

References

[1] Cristina L. Abad, Yi Lu, and Roy 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.

[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.

[3] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno M. Preguiça, and Victor Fonte. Scalable and accurate causality tracking for eventually consistent stores. In Magoutis and Pietzuch [31], pages 67–81.

[4] 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.

[5] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, 2013.

[6] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In Magoutis and Pietzuch [31], pages 126–140.

[7] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In Larry L. Peterson and Timothy Roscoe, editors, *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*. USENIX, 2006.

[8] Iwan Briquemont. Optimising client-side geo-replication with partially replicated data structures. Master's thesis, Louvain-la-Neuve, September 2014.

[9] Aimee Chanthadavong. Internet of things to drive explosion of useful data: Emc. Technical report, ZDNet, April 2014.

[10] Cisco. The zettabyte era-trends and analysis. Technical report, Cisco, June 2014.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[12] 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 [43], pages 261–264.

[13] Sean Cribbs and Russell Brown. Data structures in riak. RICON, San Francisco, CA, USA, October 2012.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[15] 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.

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

[17] 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.

[18] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 173–184, Washington, DC, USA, 2013. IEEE Computer Society.

[19] Felix Gessert, Florian Bucklers, and Norbert Ritter. Orestes: A scalable database-as-a-service architecture for low latency. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 215–222. IEEE, 2014.

[20] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[21] Sushant Goel and Rajkumar Buyya. Data replication strategies in wide area distributed systems. In Robin G. Qiu, editor, *Enterprise Service Computing: From Concept to Deployment*, pages 211–241. Idea Group Inc, 2006.

[22] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *Reliable Software Technologies – Ada Europe 96*, volume 1088 of *Lecture Notes in Computer Science*, pages 38–57. Springer Berlin Heidelberg, 1996.

[23] Junsang Kim; Won Joo Lee; Changho Jeon. A priority based adaptive data replication strategy for hierarchical cluster grids. *International Journal of Multimedia & Ubiquitous Engineering*, 9(6):127–140, 2014.

[24] Ricardo Jiménez-Peris, M. Patiño Martínez, Gustavo Alonso, and Bettina Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, September 2003.

[25] 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, February 2013.

[26] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.

[27] 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 [43], pages 265–278.

[28] 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.

[29] 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.

[30] Thanasis Loukopoulos and Ishfaq Ahmad. Static and adaptive distributed data replication using genetic algorithms. *J. Parallel Distrib. Comput.*, 64(11):1270–1285, November 2004.

[31] Kostas Magoutis and Peter Pietzuch, editors. *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8460 of *Lecture Notes in Computer Science*. Springer, 2014.

[32] 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.

[33] Prince Mahajan, Srinath T. V. Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Michael Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12, 2011.

[34] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in winfs. *Distributed Computing*, 20(3):209–219, 2007.

[35] Noriyani Mohd. Zin, A. Noraziah, AinulAzila Che Fauzi, and Tutut Herawan. Replication techniques in data grid environments. In Jeng-Shyang Pan, Shyi-Ming Chen, and NgocThanh Nguyen, editors, *Intelligent Information and Database Systems*, volume 7197 of *Lecture Notes in Computer Science*, pages 549–559. Springer Berlin Heidelberg, 2012.

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

[37] João Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. AutoPlacer: scalable self-tuning data placement in distributed key-value stores. In *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC'13, San Jose, CA, USA, June 2013. USENIX.

[38] Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, pages 288–301, 1997.

[39] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Rapport de recherche RR-7506, INRIA, January 2011. Printed.

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

[41] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400. ACM, 2011.

[42] 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.

[43] 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.

[44] K.M. Tolle, D. Tansley, and A.J.G. Hey. The fourth paradigm: Data-intensive scientific discovery [point of view]. *Proceedings of the IEEE*, 99(8):1334–1337, Aug 2011.

[45] Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Comput. Surv.*, 38(1), June 2006.

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

[47] J. von Neumann. *Automata Studies*, chapter Probabilistic Logics and Synthesis of Reliable Organisms from Unreliable Components, pages 43–98. Princeton University Press, 1956.

[48] Zhe Wang, Tao Li, Naixue Xiong, and Yi Pan. A novel dynamic network data replication scheme based on historical access record and proactive deletion. *J. Supercomput.*, 62(1):227–250, October 2012.

[49] Ouri Wolfson. A distributed algorithm for adaptive replication of data. Technical report, Department of Computer Science, Columbia University, 1990.

[50] 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. Technical Report RR-8347, August 2013.

A Making Operation-based CRDTs Operation-based

Making Operation-based CRDTs Operation-based

Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker

HASLab/INESC TEC and Universidade do Minho, Portugal

Abstract. Conflict-free Replicated Datatypes (CRDT) can simplify the design of eventually consistent systems. They can be classified into state-based or operation-based. Operation-based designs have the potential for allowing very compact solutions in both the sent messages and the object state size. Unfortunately, the current approaches are still far from this objective. In this paper, we introduce a new 'pure' operation-based framework that makes the design and the implementation of these CRDTs more simple and efficient. We show how to leverage the meta-data of the messaging middleware to design very compact CRDTs, while only disseminating operation names and their optional arguments.

1 Introduction

Eventual consistency [1] is a relaxed consistency model that is often adopted by large-scale distributed systems [2–5] where losing availability is normally not an option, whereas delayed consistency is acceptable. In eventually consistent systems, data replicas are allowed to temporarily diverge, provided that they can eventually be reconciled into a common consistent state. Reconciliation (or merging) used to be error-prone, being application-dependent, until new datatype-dependent models like the Conflict-free Replicated DataTypes (CRDTs) [6, 7] were recently introduced. CRDTs allow both researchers and practitioners to design correct replicated datatypes that are always available, and are guaranteed to eventually converge once all operations are known to all replicas. Though CRDTs have been successfully deployed in practice [2], a lot of work is still required to improve their designs and performance.

CRDTs support two complementary designs: operation-based (or simply, op-based) and state-based. In principle, op-based designs are supposed to disseminate operations, while state-based designs disseminate object states. In op-based designs [8, 7], the execution of an operation is done in two phases: *prepare* and *effect*. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the operation is applied remotely using *effect*. Different replicas are guaranteed to converge as long as messages are disseminated through a reliable causal broadcast messaging middleware, and *effect* is designed to be commutative for concurrent operations. On the other hand, in a state-based design [9, 7], an operation is only executed on the local replica state. A replica propagates its local changes to other replicas through shipping its entire state. A received state is incorporated with the local state via a *merge* function that, deterministically, reconciles the *merged* states. To maintain convergence, *merge* is defined as a join: a least upper bound over a join-semilattice [9, 7].

Typically, state-based CRDTs support ad hoc dissemination of states and can handle duplicate and out-of-order delivery of messages without breaking causal consistency; however, they impose complex state designs and store extra meta-data. On the other hand, in the systems where the message dissemination layer guarantees reliable causal broadcast, operation-based CRDTs have more advantages as they can allow for simpler implementations, concise replica state, and smaller messages. This paper only focuses on op-based CRDTs.

Unfortunately, current designs of op-based CRDTs [6] do not fully exploit the benefits that an op-based approach is supposed to offer. The relaxed framework that is currently used to build op-based CRDTs leads to many efficiency and complexity issues.

In standard op-based CRDTs the designer is given much freedom in defining *prepare*, namely using the state in an arbitrary way. This is needed to have the *effect*s of concurrently invoked data-type operations commute, and thus provide replica convergence despite the absence of causality information in current causal delivery APIs. This forces current op-based designs to include causality information in the state to be used in *prepare*, sent in messages, and subsequently used in *effect*. The designer ends up intervening in many components (the state, *prepare*, *effect*, and *query* functions) in an ad hoc way. This can result in large complex state structures and also large messages.

Currently, a *prepare* not only builds messages that duplicate the information already present in the middleware (even if it is not currently made available), but causality meta-data is often incorporated in the object state, hence, reusing design choices similar to those used in state-based approaches. Such designs, are made to work with little messaging guarantees impose larger state size, and do not fully exploit causal delivery guarantees. This freedom in current op-based designs is against the spirit of 'sending operations', and leads to confusion with the state-based approach. Indeed, in the current op-based framework, a *prepare* can return the full state, and an *effect* can do a full state-merge (which mimics a state-based CRDT) [9, 7].

We believe that the above weaknesses can be avoided if the causality meta-data can be provided by the messaging middleware. Causal broadcast implementations already posses that information internally, but it is not exposed to clients. In this paper we propose and exploit such an extended API to achieve both simplicity and efficiency in defining op-based CRDTs.

We introduce a *Pure* Op-Based CRDT framework, in which *prepare* cannot inspect the state, being limited to returning the operation (including potential parameters). The entire logic of executing the operation in each replica is delegated to *effect*, which is also made generic (i.e., not datatype dependent). For pure op-based CRDTs, we propose that the object state is a *partially ordered log* *of operations – a PO-Log* . Causality information is provided by an extended messaging API: *tagged reliable causal broadcast* (TRCB). We use this information to preserve convergence and also design compact and efficient CRDTs through a *semantically based PO-Log compaction* framework, which makes use of a datatype-specific *obsolescence* relation, defined over timestamp-operation pairs.

Furthermore, we propose an extension that improves the design and implementation of op-based CRDTs through decomposing the state into two components: a PO-Log (as before), and a causality-stripped-component which, in many cases, will be simply a standard sequential datatype. The idea is that operations are kept only transiently in the PO-Log, but once they become *causally stable*, causality meta-data is stripped, and the operations are stored in the sequential datatype. This reduces the storage overhead to a level that was never achieved before in CRDTs, neither state-based nor op-based.

2 System Model and Notations

2.1 System and Fault Models

The system is composed of a fixed set of nodes, each with a globally unique identifier in a set I. Nodes execute operations at different speeds and communicate using asynchronous message passing, abstracted by reliable causal broadcast (or gossip in the brief discussion about state-based CRDTs). Messages can be lost, reordered or duplicated, and the system can experience arbitrary, but transient, partitions. A node can fail by crashing and can recover later on; upon recovery, the last durable state of a node is assumed to be intact (not destroyed). We do not consider Byzantine faults. A fixed membership is assumed for causal broadcast: messages towards a node that is temporarily crashed or partitioned are buffered until it becomes reachable.

For presentation purposes, and without loss of generality, we consider a single object that is replicated at each node; each replica initially starts with the same state. Once a datatype operation is locally applied on a replica, the latter can diverge from the other replicas, but it may eventually convergence as new operations arrive. A local operation is applied atomically on a given replica.

2.2 Definitions and Notations

Σ denotes the type of the state. $\mathcal{P}(V)$ denotes a *power set* (the set of all subsets of V). The initial state of a replica i is denoted by $\sigma_i^0 \in \Sigma$. Operations are taken from a set O and can include arguments (in which case they are surrounded by brackets, e.g., inc and $[\mathsf{add}, v]$). We use total functions $K \rightarrow V$ and maps (partial functions) $K \hookrightarrow V$ from keys to values, both represented as sets of pairs (k, v). Given a function m, the notation $m\{k \mapsto v\}$ maps k to v, and behaves like m on other keys, e.g., Figure 1a.

$$\begin{aligned}\Sigma = I \to \mathbb{N} \quad & \sigma_i^0 = \{(r,0) \mid r \in I\} \\ \mathsf{apply}_i(\mathsf{inc}, m) &= m\{i \mapsto m(i)+1\} \\ \mathsf{eval}_i(\mathsf{rd}, m) &= \sum_{r \in I} m(r) \\ \mathsf{merge}_i(m, m') &= \{(r, \max(m(r), m'(r))) \mid r \in I\}\end{aligned}$$

(a) State-based counter

$$\begin{aligned}\Sigma = \mathbb{N} \quad & \sigma_i^0 = 0 \\ \mathsf{prepare}_i(\mathsf{inc}, n) &= \mathsf{inc} \\ \mathsf{effect}_i(\mathsf{inc}, n) &= n+1 \\ \mathsf{eval}_i(\mathsf{rd}, n) &= n\end{aligned}$$

(b) Op-based counter

Fig. 1: Counter CRDT in both state-based and op-based approaches.

2.3 Conflict-free Replicated Data Types Approaches

State-based CRDTs. These CRDTs maintain a *state* representation of an object, which evolves according to a well defined partial order. A state evolves via executing datatype operations or through applying a *join* operation, which merges any two states, thus resolving conflicting states. State-based replicas of an object converge by always shipping the entire local state, and applying the join operation on received states. State-based CRDTs are costly as the entire replica state must be shipped, but they demand less guarantees from the network because joins are designed to be commutative, idempotent, and associative. Figure 1a represents a state-based increment-only counter. In this paper we do not address state-based CRDTs.

Operation-based CRDTs. In op-based CRDTs, representations of operations issued at each node are reliably broadcast to all replicas. Once all replicas receive all issued operations (on all nodes), they eventually converge to a single state, if: (a) operations are broadcast via a reliable causal broadcast, and (b) 'applying' representations of concurrently issued operations is commutative. Op-based CRDTs can often have a simple and compact state since they can rely on the exactly-once delivery properties of the broadcast service, and thus do not have to explicitly handle non-idempotent operations. Figure 1b represents an op-based increment-only counter. The state contains a simple integer counter that is incremented for each inc operation that is delivered.

The API of the underlying middleware at each node i provides an interface method $\mathsf{cbcast}_i(m)$ that sends a message m using causal broadcast. When applying an operation o at some node i with state σ, function $\mathsf{prepare}_i(o, \sigma)$ is called returning a message m. This message is then broadcast by calling $\mathsf{cbcast}_i(m)$. Once m is delivered to each destination node j, $\mathsf{effect}_j(m, \sigma)$ is called, returning the new replica state σ'. For each node that broadcasts a given operation, the broadcast, the corresponding local delivery, and the *effect* on the local state are executed atomically. When a query operation q is performed, $\mathsf{eval}_i(q, \sigma)$ is invoked. eval takes the query and the state as input and may return a result (leaving the state unchanged).

$$
\begin{aligned}
\Sigma = \mathbb{N} \times \mathcal{P}(I \times \mathbb{N} \times V) \quad & \sigma_i^0 = (0, \{\}) \\
\mathsf{prepare}_i([\mathsf{add}, v], (n, s)) &= [\mathsf{add}, v, i, n+1] \\
\mathsf{effect}_i([\mathsf{add}, v, i', n'], (n, s)) &= (n' \textbf{ if } i = i' \textbf{ otherwise } n, s \cup \{(v, i', n')\}) \\
\mathsf{prepare}_i[\mathsf{rmv}, v], (n, s)) &= [\mathsf{rmv}, \{(v', i', n') \in s \mid v' = v\}] \\
\mathsf{effect}_i([\mathsf{rmv}, r], (n, s)) &= (n, s \setminus r) \\
\mathsf{eval}(\mathsf{rd}, (n, s)) &= \{v \mid (v, i', n') \in s\}
\end{aligned}
$$

Fig. 2: Standard op-based observed-remove add-wins set.

3 Pure Op-based CRDTs

In this section we introduce *pure op-based CRDTs* and discuss what datatypes can be implemented as pure using standard causal broadcast.

Definition 1 (Pure op-based CRDT). *An op-based CRDT is* pure *if messages contain only the operation (including arguments, if any). Given operation o and state σ,* prepare *is always defined as:*

$$\mathsf{prepare}(o, \sigma) = o.$$

This means that prepare cannot build an arbitrary message depending on the current state; in fact, in pure op-based CRDTs the operation can be immediately broadcast without even reading the replica state. As an example, the counter in Figure 1b is pure op-based, while the observed-remove set implementation (from [6]) in Figure 2 is not, because in a remove operation prepare builds a set of triples present in the current state, to be removed from the state at each replica when performing effect.

3.1 Pure Implementations of Commutative Datatypes

As we discuss now, the pure model of op-based CRDTs can be directly applied, using standard reliable causal broadcast [10], to implement datatypes whose operations are commutative.

Definition 2 (Commutative datatype). *A concurrent datatype is commutative if (a) for any operations f and g, their (sequential) invocation commutes: $f(g(\sigma)) = g(f(\sigma))$, and (b) concurrent invocations are defined as equivalent to some linearization.*

Commutative datatypes reflect a *principle of permutation equivalence* [11] stating that "If all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states".

As the extension to concurrent scenarios follows directly from their sequential definition, with no room for design choices, commutative datatypes can have a

$$\Sigma = \mathbb{N} \quad \sigma_i^0 = 0$$
$$\mathsf{prepare}_i(o, \sigma) = o$$
$$\mathsf{effect}_i(\mathsf{inc}, n) = n + 1$$
$$\mathsf{effect}_i(\mathsf{dec}, n) = n - 1$$
$$\mathsf{eval}_i(\mathsf{rd}, n) = n$$

(a) Pure PN-counter

$$\Sigma = \mathcal{P}(V) \quad \sigma_i^0 = \{\}$$
$$\mathsf{prepare}_i(o, \sigma) = o$$
$$\mathsf{effect}_i([\mathsf{add}, v], s) = s \cup \{v\}$$
$$\mathsf{eval}_i(\mathsf{rd}, s) = s$$

(b) Pure grow-only set

Fig. 3: Pure op-based CRDTs for commutative datatypes.

standard sequential specification and implementation. As such, a pure op-based CRDT implementation is trivial: as when using the standard causal broadcast, the message returned from prepare, containing the operation, will arrive exactly once at each replica, it is enough to make effect consist simply in applying the received operation to the state, over a standard sequential datatype, i.e., defining for any datatype operation o:

$$\mathsf{effect}_i(o, \sigma) = o(\sigma).$$

Two examples of commutative datatypes, presented in Figure 3, are: a PN-counter with inc and dec operations; a grow-only set (G-set) with add operation. Both cases use a standard sequential datatype for the replica state, and applying effect is just invoking the corresponding operation in the sequential datatype. Both these examples explore commutativity and rely on the exactly-once delivery, leading to a trivial pure implementation.

3.2 Non-commutative Datatypes

In the case where datatype operations are not commutative, such as a set with add and rmv operations, where $\mathsf{add}(v, \mathsf{rmv}(v, s)) \neq \mathsf{rmv}(v, \mathsf{add}(v, s))$, we have two reasons that prevent effect from being simply applying the operation over a sequential datatype.

One reason is that, even when the semantics of concurrent invocations can be defined as equivalent to some linearization of those operations, the messages corresponding to concurrent operations will be, in general, delivered in different orders in different replicas. Therefore, as the operations do not commute, simply applying them in different orders in different replicas makes replicas diverge. Under the assumption of causal delivery and the aim of convergence, effect must always be commutative, and therefore, cannot be defined directly as operations that are not commutative themselves. It must be defined in some other way.

The other reason is that it is useful to specify concurrent datatypes in which the outcomes of concurrent executions are not equivalent to some linearization. The best example is the multi-value register, where two concurrent writes make a read in their causal future return a set with both values written. This outcome could not arise under a sequential specification.

```
state:
    $\sigma_i \in \Sigma$
on $\mathsf{operation}_i(o)$:
    $\mathsf{tcbcast}_i(\mathsf{prepare}(o, \sigma_i))$
on $\mathsf{tcdeliver}_i(m, t)$:
    $\sigma_i := \mathsf{effect}(m, t, \sigma_i)$
on $\mathsf{tcstable}_i(t)$:
    $\sigma_i := \mathsf{stable}(t, \sigma_i)$
```

Algorithm 1: Distributed algorithm for node i using tagged causal broadcast.

In general, a concurrent datatype will have a specification depending on the partial order of operations over the datatype. Given that such information about that partial order is already present in metadata in causal delivery middleware, we propose an approach for pure op-based CRDTs for general non-commutative datatypes that leverages this metadata, now exposed by an extended causal delivery API, what we call *tagged reliable causal broadcast*.

4 Tagged Reliable Causal Broadcast (TRCB)

A common implementation strategy for a reliable causal broadcast service [12] is to assign a vector clock to each message broadcast and use the causality information in the vector clock to decide at each destination when a message can be delivered. If a message arrives at a given destination before causally preceding messages have been delivered, the service delays delivery of that message until those messages arrive and are delivered. Unlike totally ordered broadcast, which requires a global consensus on the delivery order, causal broadcast can progress with local decisions. For general datatypes, causal consistency is likely the strongest consistency criteria compatible with an always-available system that eventually converges [13].

By leveraging this information, we can specify a reliable causal broadcast service with an extended API, and refer to its broadcast operation at each replica i as $\mathsf{tcbcast}_i(m)$. Algorithm 1, running on each node i, shows how the events triggered by the tagged causal delivery service are used to invoke the generic functions for pure op-based CRDTs: prepare, effect and stable; these functions, in different variants, will be discussed in the following sections. This extended service provides nodes with information about two aspects.

Partial order The first salient difference is that message delivery on each node i, given by the event $\mathsf{tcdeliver}_i(m, t)$, provides not only the message m itself, but also the vector clock timestamp t corresponding to m. When implementing pure op-based CRDTs, in which only the operations are sent in messages, we can use the timestamp supplied by the service upon delivery in the definition of effect; i.e., we can have $\mathsf{effect}(o, t, s)$ as a function of the operation, the timestamp and the current state.

As we will see in the next section, this information about the partial order can be embedded in the state in a general way so that effect is commutative and reference implementations of general possibly non-commutative datatypes can be obtained, following their specification. Moreover, in Section 6 we will see how realistic efficient pure CRDTs can be obtained, in which the use of this causality information, together with the semantics of the datatype operations is essential.

Causal stability TRCB also provides information about what we denote by *causal stability.*

Definition 3 (Causal Stability). *A clock t, and corresponding message, is* causally stable *at node i when all messages subsequently delivered at i will have timestamp $t' \geq t$;*

This implies that no message with a timestamp t' concurrent with t can be delivered at i when t is causally stable at i. This notion differs from classic message stability [10] in which a message is stable if it has been delivered at all nodes. Here we not only need this to happen but also that no further concurrent messages may be delivered. Therefore, causal stability is a stronger notion, implying classic message stability.

The extended API will provide an event $\mathsf{tcstable}_i(t)$ which will be triggered when it is determined that t is stable at i. The middleware at node i can check if timestamp t is causally stable at i by checking if a message with timestamp $t' \geq t$ has already been delivered at i from every other other node j, i.e.:

$$\mathsf{tcstable}_i(t) \ \textbf{when} \ \forall j \in I \setminus \{i\} \cdot \exists t' \in \mathsf{delivered}_i() \cdot \mathsf{origin}(t') = j \wedge t \leq t',$$

where $\mathsf{delivered}_i()$ returns the set of messages that have been delivered at node i, while $\mathsf{origin}(t)$ denotes the node from where the message corresponding to t has been sent. To evaluate this clause efficiently, the middleware only needs to keep the most recently delivered timestamp from each origin [14].

We will see in Section 6 how causal stability can be used to reduce CRDT state size, by stripping causality information from causally stable operations.

5 Pure CRDTs Based on a Partially Ordered Log

Having a tagged causal broadcast service, it is now possible to obtain a universal mechanism for obtaining pure reference implementations for any (possibly non-commutative) concurrent datatype in which semantics are defined over the partial order of operations.

The reference mechanism, presented in Figure 4, uses a uniform notion of state for a replica: a *partially ordered log of operations*, what we call a PO-Log. This uses the ordering information offered by the messaging middleware to keep information about concurrent operations, not trying to impose a local total-order over them, contrary to a classic sequential log.

The PO-Log can be defined as a map (a partial function) $T \hookrightarrow O$ from message timestamps (as given by the tagged causal broadcast service) to the

$$\begin{aligned} \Sigma = T \hookrightarrow O \quad & \sigma_i^0 = \{\} \\ \mathsf{prepare}(o, s) &= o \\ \mathsf{effect}(o, t, s) &= s \cup \{(t, o)\} \\ \mathsf{eval}(q, s) &= \text{[datatype-specific query function over partial order]} \end{aligned}$$

Fig. 4: PO-Log based reference implementation for pure op-based CRDTs.

$$\begin{aligned} \Sigma = T \hookrightarrow O \quad & \sigma_i^0 = \{\} \\ \mathsf{prepare}(o, s) &= o \qquad \qquad \text{(with } o \text{ either } [\mathsf{add}, v] \text{ or } [\mathsf{rmv}, v]) \\ \mathsf{effect}(o, t, s) &= s \cup \{(t, o)\} \\ \mathsf{eval}(\mathsf{rd}, s) &= \{v \mid (t, [\mathsf{add}, v]) \in s \land \nexists (t', [\mathsf{rmv}, v]) \in s \cdot t < t'\} \end{aligned}$$

Fig. 5: PO-Log based observed-remove add-wins set.

corresponding operation. Here we have a universal datatype-independent definition of effect as:

$$\mathsf{effect}(o, t, s) = s \cup \{(t, o)\},$$

which is trivially commutative, as needed. Only the query functions will need datatype-specific definitions according to desired semantics. Their definition over the PO-Log will typically be a direct transposition of their specification.

Figure 5 shows a pure PO-Log based implementation of an *add-wins* observed-remove set over the new tcbcast service. The *add-wins* semantic is defined in the rd query function: the set of values reported to be in the set are those values that have been added with no rmv causally in the future of the add. Another example, shown in Figure 6, is a multi-value register. Here a read reports the set of all concurrently written values that have not been subsequently overwritten.

These reference implementations are not realistic to be actually used, namely because state size in each replica is linear with the number of operations. They are a starting point from which actual efficient implementations can be derived, by semantically based PO-Log compaction, as we show in the next section. But they are relevant, as they provide a clear description of the concurrent semantics of the replicated datatype. This is possible since we are capturing the partial ordered set of all operations delivered to each replica. (A similar approach to express the semantics is found in [15] when relating to the *visibility* relation.)

6 Semantically Based PO-Log Compaction

We now show how PO-Log based CRDTs can be made efficient by performing PO-Log compaction. There are two ingredients that we explore. The first one is to prune the PO-Log after each operation is delivered in the effect, so as to keep the minimum number of operations such that all queries return the same result

$$\begin{aligned}
\Sigma = T \hookrightarrow O \quad & \sigma_i^0 = \{\} \\
\mathsf{prepare}([\mathsf{wr}, v], s) &= [\mathsf{wr}, v] \\
\mathsf{effect}([\mathsf{wr}, v], t, s) &= s \cup \{(t, [\mathsf{wr}, v])\} \\
\mathsf{eval}(\mathsf{rd}, s) &= \{v \mid (t, [\mathsf{wr}, v]) \in s \wedge \nexists (t', [\mathsf{wr}, v']) \in s \cdot t < t'\}
\end{aligned}$$

Fig. 6: PO-Log based multi-value register.

$$\begin{aligned}
\Sigma = T \hookrightarrow O \quad & \sigma_i^0 = \{\} \\
\mathsf{prepare}(o, s) &= o \\
\mathsf{effect}(o, t, s) &= \{x \in s \mid \neg\, \mathsf{obsolete}(x, (t, o))\} \cup \{(t, o) \mid x \in s \Rightarrow \neg\, \mathsf{obsolete}((t, o), x)\} \\
\mathsf{obsolete}() &= [\text{datatype-specific relation to identify obsolete operations}] \\
\mathsf{eval}(q, s) &= [\text{datatype-specific query function over partial order}]
\end{aligned}$$

Fig. 7: Reference implementation for PO-Log compaction.

as when the full PO-Log is present. The second one is to explore causal stability information, to discard timestamp information for elements once they become stable, possibly merging some elements.

6.1 Exploring Causality Information

As the possibility of discarding operations while preserving semantics is datatype dependent, we propose a unified framework which includes the PO-Log, prepare, and a more sofisticated effect which makes use of a datatype-specific relation to discard operations made irrelevant by newer arrivals, according to both operation content and corresponding timestamp, as shown in Figure 7.

This relation between pairs timestamp-operation – $\mathsf{obsolete}((t, o), (t', o'))$ – is used by effect in the following way: when a new pair (t, o) is delivered to a replica, effect discards from the PO-Log all elements x such that $\mathsf{obsolete}(x, (t, o))$ holds; also, the delivered pair (t, o) is only inserted into the PO-Log if it is not redundant itself, according to the current elements, i.e., if for any current x in the PO-Log $\mathsf{obsolete}((t, o), x)$ is false. This relation is not restricted to be a partial-order, but can be a more general relation, allowing, e.g., a newly arrived operation to discard others in the PO-Log without necessarily being itself added.

It is easy to see by simple induction that this execution mechanism provides the invariant that for any two different pairs p_1 and p_2 in the PO-Log, $\mathsf{obsolete}(p_1, p_2)$ is false. This invariant allows reasoning about the datatype, namely to be able to write simplified query functions that give the same result over the compact PO-Log as the original query functions over the full PO-Log.

An observed-remove add-wins set using the PO-Log compaction framework can be seen in Figure 8 (which presents only the datatype-specific funtions). Here it can be clearly seen that obsolete was defined directly according to the essence

$$\begin{aligned}
\mathsf{obsolete}((t, [\mathsf{add}, v]), (t', [\mathsf{add}, v'])) &= t < t' \wedge v = v' \\
\mathsf{obsolete}((t, [\mathsf{add}, v]), (t', [\mathsf{rmv}, v'])) &= t < t' \wedge v = v' \\
\mathsf{obsolete}((t, [\mathsf{rmv}, v]), x) &= \text{true} \\
\mathsf{eval}(\mathsf{rd}, s) &= \{v \mid (t, [\mathsf{add}, v]) \in s\}
\end{aligned}$$

Fig. 8: Observed-remove add-wins set with PO-Log compaction.

$$\begin{aligned}
\mathsf{obsolete}((t, [\mathsf{wr}, v]), (t', [\mathsf{wr}, v'])) &= t < t' \\
\mathsf{eval}(\mathsf{rd}, s) &= \{v \mid (t, [\mathsf{wr}, v]) \in s\}
\end{aligned}$$

Fig. 9: Multi-value registed with PO-Log compaction.

of the datatype: a subsequent add obsoletes a previous add of the same value; a rmv obsoletes an add of the same value. The more interesting rule is that any rmv is made obsolete by any other timestamp-operation pair; this implies that a rmv can only exist as the single element of a PO-Log (if it was inserted into an empty PO-Log), being discarded once other operation arrives (including another rmv), and never being inserted into a non-empty PO-Log. This reflects the *add-wins* nature, in which the role of a rmv is basically to discard same-value additions in its causal past. Under the now compacted PO-Log, the query function rd can be defined in a simple way, and clearly seen to give the same result as the original one over the full PO-Log (in Figure 5).

Another example where PO-Log compaction leads to an efficient datatype is the multi-value register, in Figure 9, where it is obvious the effect of a write in making all writes in its causal past obsolete, regardless of value written. The set of concurrent writes that have not been made obsolete will be returned in a read, which is equivalent to the original definition in Figure 6 over the full PO-Log.

6.2 Exploring Causal Stability Information

The second component of PO-Log compaction involves using causal stability to strip logical clocks from the PO-Log. From the definition of causal stability, if some pair (t, o) is in the PO-Log, with t causally stable, all future deliveries (t', o') used in effect will be causally in the future, i.e., $t' > t$.

Because effect only compares, through obsolete, new arrivals and PO-Log elements – but never PO-Log elements among themselves – and for a stable t, all future deliveries will be causally in its future, the value of t is no longer needed, and it can be replaced by any timestamp that is less than all other timestamps: the bottom $(\perp)$ of the timestamp domain; e.g., a null map $\{\}$ for a vector-clock timestamp.

In practice this means that, instead of having timestamps that are maps/vectors with size linear on the number of replicas, we can have a special marker denoting

$$\begin{aligned}
\Sigma = \mathcal{P}(O) \times (T \hookrightarrow O) \quad & \sigma_i^0 = (\{\}, \{\}) \\
\mathsf{prepare}(o, (s,p)) &= o \\
\mathsf{effect}(o, t, (s,p)) &= (s', p'), \text{ where} \\
s' &= \{x \in s \mid \neg\, \mathsf{obsolete}((\bot, x), (t, o))\} \\
p' &= \{x \in p \mid \neg\, \mathsf{obsolete}(x, (t, o))\} \cup \{(t, o) \mid x \in p \Rightarrow \neg\, \mathsf{obsolete}((t,o), x)\}, \\
\mathsf{stable}(t, (s,p)) &= (s \cup p(t), p \setminus \{(t, p(t))\}) \\
\mathsf{obsolete}() &= [\text{datatype-specific relation to identify obsolete operations}] \\
\mathsf{eval}(q, (s,p)) &= [\text{datatype-specific query function over PO-Log }]
\end{aligned}$$

Fig. 10: Full PO-Log compaction framework exploring causal stability.

bottom (e.g. a null pointer). This greatly diminishes the size of a replica state for two reasons. For some CRDTs where size may be a problem, like sets of integers, values may take considerably less space than timestamps, which constitute the great percentage of state size; stripping a timestamp from an element can be a huge percentual improvement per element. The second reason is that, again in such scenarios with large states, the percentage of elements in the PO-Log that are not yet causally stable will be quite small, with most already stable. This means that this optimization, which gives good results per element, will be applied to most elements, leading to a large overall improvement.

Instead of being a map $T \hookrightarrow O$, the PO-Log is split in $\mathcal{P}(O) \times (T \hookrightarrow O)$, detaching into a plain set all operations that have stable timestamps (i.e., making the $\bot$ timestamp implicit). A new framework using the split PO-Log is presented in Figure 10. In effect a new delivery possibly discards elements both from the set of operations and the partially ordered set of operations; elements from the latter are used to decide on the addition of the new delivery, as before. (The possibility of a stable operation obsoleting a new arrival, in its causal future, is not considered, but this can easily be changed if some example shows its usefulness.) In stable, the operation corresponding to a stable timestamp is fetched, to be added to the set and the corresponding entry removed from the map.

This PO-Log split allows actual implementations, tailored to specific datatypes to achieve further improvements. As an example, an or-set will have only $(t, [\mathsf{add}, v])$ entries in its PO-Log (except for the singleton $\{(t, [\mathsf{rmv}, v]\})$ which can be prevented by a special rule); when elements become causally stable, only the value v needs to be migrated and not the operation name. The set component of the split PO-Log becomes a plain set of values (and not lists of operation-argument), allowing traditional implementations of sets to be used, for example a bitmap if the datatype is for a dense set of integers. (Such implementations will need specialized versions of effect to avoid the implicit traversal of the set when applying obsolete, but that is something desirable in any actual implementation.)

By exploiting both causality and causal stability information, made available by the proposed tagged causal delivery API, we have paved the way for these optimizations that allow pure op-based CRDTs that are much more suitable for large datatypes than current designs.

7 Related Work

Weakly consistent replication The design of replicated systems that are always available and eventually converge can be traced back to historical designs in [14, 16, 17], among others. Lazy Replication [18] allows enforcing causal consistency, but may apply concurrent operations in different order in different replicas, possibly leading to divergence if operations are not commutative; TSAE [19] also either applies concurrent operations in possibly different orders, or allows enforcing a total order compatible with causality, at the cost of delaying message delivery. Both these systems use a message log, the former with complete causality information, but the log is *pre-delivery*, unseen by the application: operations are applied sequentially to the current state and queries use only the state. In our framework the PO-Log is *post-delivery*, being part of the datatype state, maintains causality information and is used in query operations.

Conflict-free replicated data types The formalization of the commutativity requirements for concurrent operations in replicated datatypes was introduced in [8, 20], and that of the state based semi-lattices was presented in [9]. Afterwards, the integration of the two models with many extensions was presented in Conflict-free Replicated Datatypes [6, 7]. Currently, CRDTs have made their way into the industry through designing highly available scalable systems in cloud databases like RIAK [2], and mobile gaming industry such as Rovio [21].

Message stability The notion of message stability was defined in [10] to represent a message that has been received by all recipients; each replica can discard any message it knows to be stable after delivering it. Similar notions are used in Lazy Replication [18] and TSAE [19]. In all these cases the aim is message garbage collection. Our definition of *causal stability* is the stronger notion that *no more concurrent messages will be delivered*; here we use it inside the datatype both to discard operations acting as tombstones, and to discard causality information while keeping the operation. Causal stability is close to what is used in the mechanics of Replicated Growable Arrays (RGA) [20], although no definition is presented there.

Message obsolescence Semantically reliable multicast [22] uses the concept of *message obsolescence* to avoid delivering messages made redundant by some newly arrived message, where obsolescence is a strict partial order that is a subset of causality, possibly relating messages from the same sender or totally ordered messsages from different senders. Our obsolescence relation is more general, being defined on clock-operation pairs, and can relate concurrent messages. Also, it is defined per-datatype, being used inside each datatype, post-delivery.

8 Conclusions

In this paper we improved the CRDT model by introducing the stricter notion of *pure* op-based, and establishing a clear frontier with state-based models, breaking the equivalence between the two models for general datatypes. We have shown which pure datatypes are possible over off-the-shelf causal delivery middleware and then introduce an extended API, *tagged* reliable broadcast, that supports the remaining datatypes, those non-commutative in their sequential specifications. Supported by this API, that conveys causal information present in the middleware, we were able to define a partially ordered log, named PO-Log, that supports a clear semantic description and abstract implementation of each concurrent datatype.

To obtain efficient implementations we developed a framework for semantic compaction of a PO-Log and, in a final step, resorted to a notion of *causal stability* to determine when it is safe to strip PO-Log entries of their causal order metadata. This final step allows eventually moving all data to a standard sequential datatype, or a local database, and harvest the efficiency gains of reusing existing optimized data structures and database engines.

Having exemplified the framework with relevant non-trivial datatypes (replicated sets and registers) we expect that future research, and the existing developer community, can apply these techniques to other derived datatypes, such a maps, graphs, and sequences.

Aknowledgments We would like to thank Marek Zawirski, Ricardo Gonçalves and anonymous reviewers for comments that helped improve this work. Project Norte-01-0124-FEDER-000058 is co-financed by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). Funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 609551, SyncFree project.

References

1. Vogels, W.: Eventually consistent. ACM Queue **6**(6) (October 2008) 14–19
2. Cribbs, S., Brown, R.: Data structures in Riak. In: Riak Conference (RICON), San Francisco, CA, USA (oct 2012)
3. Bailis, P., Ghodsi, A.: Eventual consistency today: Limitations, extensions, and beyond. Queue **11**(3) (March 2013) 20:20–20:32
4. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Symp. on Op. Sys. Principles (SOSP), Copper Mountain, CO, USA, ACM SIGOPS, ACM Press (December 1995) 172–182
5. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles (SOSP). Volume 41 of Operating Systems Review., Stevenson, Washington, USA, Assoc. for Computing Machinery (October 2007) 205–220

6. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (January 2011)
7. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In Défago, X., Petit, F., Villain, V., eds.: Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS). Volume 6976 of Lecture Notes in Comp. Sc., Grenoble, France, Springer-Verlag (October 2011) 386–400
8. Letia, M., Preguiça, N., Shapiro, M.: CRDTs: Consistency without concurrency control. Rapp. Rech. RR-6956, Institut National de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France (June 2009)
9. Baquero, C., Moura, F.: Using structural characteristics for autonomous operation. Operating Systems Review **33**(4) (1999) 90–96
10. Birman, K., Schiper, A., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. **9**(3) (August 1991) 272–314
11. Bieniusa, A., Zawirski, M., Preguiça, N., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: Brief announcement: Semantics of eventually consistent replicated sets. In Aguilera, M.K., ed.: Int. Symp. on Dist. Comp. (DISC). Volume 7611 of Lecture Notes in Comp. Sc., Salvador, Bahia, Brazil, Springer-Verlag (October 2012) 441–442
12. Schmuck, F.B.: The use of efficient broadcast protocols in asynchronous distributed systems. Technical Report TR 88-928, Cornell University (1988)
13. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA (2011)
14. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC), Vancouver, BC, Canada (August 1984) 233–242
15. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. [23] 271–284
16. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (January 1976)
17. Quarterman, J.S., Hoskins, J.C.: Notable computer networks. Commun. ACM **29**(10) (October 1986) 932–971
18. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. **10**(4) (November 1992) 360–391
19. Golding, R.A.: Weak-consistency group communication and membership. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA (December 1992) Tech. Report no. UCSC-CRL-92-52.
20. Roh, H.G., Jeon, M., Kim, J.S., Lee, J.: Replicated Abstract Data Types: Building blocks for collaborative applications. Journal of Parallel and Dist. Comp. **71**(3) (March 2011) 354–368
21. Rovio Entertainment Ltd.: Rovio gaming. http://www.rovio.com/en (2013)
22. Pereira, J., Rodrigues, L., Oliveira, R.: Semantically reliable multicast: Definition, implementation, and performance evaluation. IEEE Trans. Comput. **52**(2) (February 2003) 150–165
23. Jagannathan, S., Sewell, P., eds.: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. In Jagannathan, S., Sewell, P., eds.: POPL, ACM (2014)

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

This article is currently under submission with double-blind review. Brie is the codeword for SwiftCloud, in order to make the submission anonymous.

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

EuroSys 2015 Paper #174 Total length: 14 pages

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 Brie distributed object database, which supports high numbers of client-side partial replicas. Brie 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 Brie 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. Existing client-side systems either make only limited consistency guarantees, or do not scale to large numbers of client devices, or both. App developers may resort to implementing their own ad-hoc application-level cache, 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 [36]. Recent application frameworks such as Google Drive Realtime API [14], TouchDevelop [12] or Mobius [15] 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, 6, 19, 25, 26] or for managing database replicas on clients [10, 28] 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 responsible for ensuring correct and scalable database access to client-side applications. It should address the (somewhat conflicting) requirements of consistency, availability, and convergence [27], at least as well as geo-replication 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 possible consistency model is *causal consistency* where concurrent updates to objects *converge* [25, 27].

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. Full replication at high numbers of resource-poor devices would be unacceptable [10]; but partial replication of data and metadata could cause anomalous message delivery or unavailability. It is not possible to assume, like many previous systems, that fault tolerance or consistency is solved by locating the application is located inside the data centre (DC), or has a sticky session to a single DC [7, 36].

This work addresses these challenges. We present the algorithms, design, and evaluation of Brie, 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, Brie 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. The novel aspects of our approach include:

Cloud-backed support for partial replicas (§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 it does not have a full replica, the client 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 tradeoff between staleness and unavailability. A client observes a remote update only if it is stored in some number $K \geq 1$ of DCs [28]. 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 (§4) Thanks to funnelling communication through DCs and to "reading in the past," our metadata design decouples *tracking causality*, which uses small vectors assigned in the background by DCs, from *unique identification*, based on client-assigned scalar timestamps. This ensures that the size of

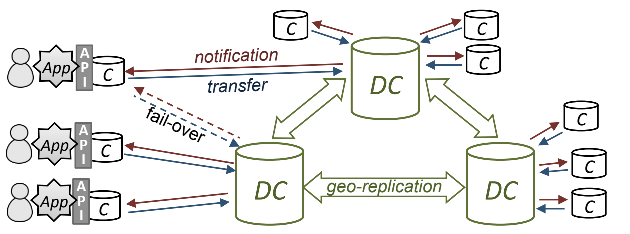


Figure 1. System components (Application processes, Clients, Data Centres), and their interfaces.

metadata is small and bounded. Furthermore, a DC can prune its log independently of clients, ensuring safety by storing a local summary of delivered updates.

We implement Brie and demonstrate experimentally that our design reaches its objective, at a modest staleness cost. We evaluate Brie in Amazon EC2, against a port of WaltSocial [35] and against YCSB [16]. When data is cached, response time is two orders of magnitude lower than for server-based protocols with similar availability guarantees. With three DC 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.

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.

Fig. 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. [35], 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 and recover with its persistent memory intact.

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.

[1] We refer to prior work for the somewhat orthogonal issues of parallelism and fault-tolerance within a DC [5, 19, 25, 26].

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 [32]. 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 [21, 27].

Causal consistency The strongest available and convergent model is causal consistency [3, 27].

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*.[2] Specifically, 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. The system must detect such a gap, and wait until u is delivered before returning y, or avoid it in the first place. Otherwise, reads with a causal gap expose both application programmers and users to anomalies [25, 26].

We consider a transactional variant of causal consistency to support 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 [8, 25, 26].

Convergence Another requirement is **convergence**, which consists of two properties: *(i)* **At-least-once delivery** (liveness): 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** (safety): two nodes that delivered the same set of updates read the same value.

Causal consistency is not sufficient to guarantee confluence, as two replicas might receive the same updates in different orders. Therefore, we rely on CRDTs, high-level

[2] This subsumes the well-known session guarantees [13].

data types that guarantee confluence and have rich semantics [13, 34]. An update on a high-level object is not just an assignment, but is a high-level 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, 18, 22], but also higher level types such as Sets, Lists, Maps, Graphs, Counters, etc. [2, 33–35]. 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. As high-level updates are often not idempotent (consider for instance increment()), safety also demands **at-most-once delivery**.

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 novel challenge.

2.2 Metadata design

Metadata serves to identify updates and to ensure correctness. 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 [25] grow with the number of updates [19, 26, §3.3], whereas version vectors [10, 28] 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, 19, 26]. 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. 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 (fail-over). 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 considerably [4, 13, 26]. 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 [10]. Consider the following example: Alice posts a photo on her wall (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 does not cache Charles' inbox, it cannot observe the causal chain $a \to b \to c$ 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.[3]

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 [10, 28].

3. The Brie approach

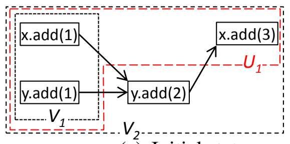
We now describe a design that addresses the above challenges, first in the failure-free case, and next, how we support DC failure.

3.1 Causal consistency at full DC replicas

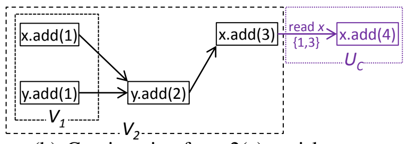
Ensuring causal consistency at fully-replicated DCs is a well-known problem [3, 19, 25, 26]. Our design is a hybrid between state-based (storing and transmitting a whole object states, called **checkpoint**) and log-based (sending and transmitting operations incrementally) [10, 30]. Hereafter, we focus on the log-based angle, and discuss checkpoints only where relevant.

A **database version**, noted U, is any subset of updates, ordered by causality. A version maps object identifiers to values (via the read API), by applying the relevant subsequence of the log. We say that a version U has a **causal gap**, or is **inconsistent** if it is not causally-closed, i.e., if $\exists \mathsf{u}, \mathsf{u}' : \mathsf{u} \to \mathsf{u}' \wedge \mathsf{u} \notin U \wedge \mathsf{u}' \in U$. 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

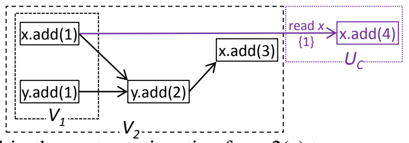
[3] Note that David can still perform updates, but they cannot be delivered. From David's perspective, writes remain available. However, the system as a whole does not converge.



(a) Initial state



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



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

Figure 2. Example evolution of states 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.

conundrum, we adopt the approach of "reading in the past" [3, 25]. Thus, a DC exposes a gapless but possibly delayed state, noted V.

To illustrate, consider the example of Fig. 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 $[x \mapsto \{1\}, y \mapsto \{1\}]$ and V_2 with value $[x \mapsto \{1,3\}, y \mapsto \{1,2\}]$ are both gapless. However, version U_1, with value $[x \mapsto \{1,3\}, y \mapsto \{1\}]$ 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, $x = \{1,3\}$. 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 $x = \{1\}$. Then, the second read will be satisfied immediately with $y = \{1\}$. 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 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., u.$deps \subseteq V$, then it applies u. The new state is $V' = V \oplus \{u\}$, where we note by $\oplus$ a **log merge operator** that filters out duplicates, further discussed in 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 [10]. To avoid the complexity, we leverage the DC's full replicas to manage 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 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 is maintained up to date by the DC, in the background.

3.3 Failing over: the issue with transitive 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 DC availability and for performance [9]), which may create a causal gap in the client.

To illustrate the problem, return to the example of Fig. 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 $\mathsf{u} = \mathsf{add}(4)$, transitioning to the state shown in Fig. 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 to avoid creating the gap in state $V_1 \oplus U_C$.

3.3.1 Conservative read: possibly stale, but safe

To avoid such gaps that cannot be satisfied, the insight is to depend on updates that are likely to be present in the failover DC, called K**-stable** updates.

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. failures model. 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., $V_C = V_i^K|_O \oplus U_C|_O$, to support failover. In this way, the client depends, either on external updates that are likely to be found in any DC (V_i^K), or internal ones, which the client can always transfer to the new DC (U_C).

To illustrate, let us return to Fig. 2(a), and consider the conservative progression to Fig. 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 §3.2, a DC version V_i^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. Decreasing it has immediate effect without impacting correctness. Increasing K has effect only for future updates, in order to not violate montonicity.

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, some 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 [17, 25, 35] 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;[4] in the limit, $K = N$ ensures complete availability, but no client can transfer 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 §3.2, and is similar to previous blocking session-guarantee protocols [36].

$K = 2$ is a good compromise for deplyoments with three or more DCs that covers common scenarios of a DC failure or disconnection [17, 23]. 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 updates until the partition heals. To improve liveness in this scenario, Brie 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.

4. Implementation

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

4.1 Timestamps, vectors and log merge

The Brie 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 propose a new type of metadata, which fulfils the requirements and has a low cost. It

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

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. The metadata assigned to some update 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 delivery to multiple DCs (on failover, §3.3.1). The updates in a transaction all have the same timestamp(s), to ensure all-or-nothing delivery [35]. Our approach provides the flexibility to refer to an update via any of its timestamps, which is handy during failover.

We represent a version or a dependency as a **version vector** [29]. 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 u, it means that u causally depends on $\{(DC_1, 1), (DC_2, 1), (DC_2, 2)\}$. In Brie 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 (defined for states U that cover all timestamps of VV) selects every update in state U that matches the vector:

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

For the purpose of the decoding function $\mathcal{V}$, a given update can be referred equivalently through any of its timestamps. Moreover, $\mathcal{V}$ is stable with growing state 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 Brie by following the lifetime of an update, and with reference to the names in Fig. 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 $V_{DC} = \mathcal{V}(VV_{DC}, U_{DC})$.

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 §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 τ at client C, the client replica 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 base version can be updated concurrently with the transaction execution. A read in transaction τ 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, $o \notin O$, the clients extends its interest set, and returns the value once the DC updates the base version projection.

When the application issues internal update u, it is appended to the transaction buffer $\tau.U \leftarrow \tau.U \oplus \{\mathsf{u}\}$, and included in any later read. The transaction commits locally at the client and never fails [26].[5] 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 is now free to 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 u from the log. If any of 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 u to its current DC. The client expects to receive an acknowledgement from the DC, containing the timestamp T that the DC assigned to update u. If so, the client records the DC timestamp(s) in the original update record $\mathsf{u}.T_{\mathcal{DC}} \leftarrow T$.

It may now iterate with 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.

[5] To simplify the notation, and without loss of generality, we assume hereafter that a transaction performs at most one update. This is easily extended to multiple updates, by assigning the same timestamp to all the updates of the same transaction, ensuring the all-or-nothing property [35].

(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 §3.3.1 avoids repeated failures.

Upon receiving update u, the DC verifies if it 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, 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 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, $\mathsf{u}.T_{\mathcal{DC}}$. In either case, the DC acknowledges the transfer to the client with the DC timestamp(s). Note that steps (1)–(2) can be parallelised between transfer requests received from different client replicas.

Geo-replication protocol: DC to DC The geo-replication protocol consists of 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. 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 computed 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 of the client, i.e., if $\mathcal{T}(VV_{DC}') \subseteq \mathcal{T}(VV_{DC})$. Otherwise, the DC would cause a causal gap with the client's state; in this case, the client is redirected to another DC (see §3.3.1).

The DC sends over each channel a causal stream of update notifications.[6] Notifications are batched according to either time or to rate [10]. A notification packet consists of a new base version vector VV_{DC}, and a sequence of log of all the updates U_Δ to the objects of the interest set, between the client's previous base vector VV_{DC}' and the new one. Formally, $U_\Delta = \{\mathsf{u} \in U_{DC}|_O \mid \mathsf{u}.T_{\mathcal{DC}} \cap (\mathcal{T}(VV_{DC}) \setminus \mathcal{T}(VV_{DC}')) \neq \emptyset\}$. 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.

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 o, the DC responds with the current version of o, which includes all updates to o up to the base version vector. To avoid races, a notification includes a hash of the interest set, which the client checks.

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** prediocially replaces the prefix of a log and 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)* unique timestamp identification of each update, which serves to filter out duplicates by $\oplus$ operator, as explained earlier, and *(b)* the capability to compute different versions, for application processes reading at different causal times. Update u is expendable once all of its duplicates have been filtered out, and once u has been delivered to all interested application processes. However, evaluating expendability precisely would require access to the client replica states.

In practice, we need to prune *aggressively*, but still avoid the above issues, as we explain next.

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_i^{Δ}, where Δ is a real-time delay [25, 26].

To avoid duplicates, we extend our DC local metadata as follows. DC_i maintains an **at-most-once guard** G_i, which

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

	YCSB [16]	**SocialApp** [35]
Type of objects	LWW Map	Set, Counter, Register
Object payload	10×100 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	400,000 objects
Object popularity	uniform / zipfian	uniform
Session locality	40% (low) / 80% (high)	

Table 1. Characteristics of applications/workloads.

records the sequence number of each client's last pruned update $G_i : \mathcal{C} \rightarrow \mathbb{N}$. Whenever the DC receives a transfer request for update u with timestamp $(C, k) = \mathsf{u}.t_{\mathcal{C}}$ and cannot find it in its log, it checks the at-most-once guard entry whether u is contained in the checkpoint. If the update was already pruned away ($G_i(C) \geq k$), the update is ignored; the DC discarded information about the exact set of update's DC timestamps in this case; therefore, in transfer reply, they are overapproximated by vector VV_i^{Δ}. Similarly, on a client cache miss, the DC sends object state that consists of the most recent checkpoint of the object together the client's guard entry, so that the client can merge it with his updates safely. Note that a guard is local to and shared at a DC. It is *never* fully transmitted.

4.3.2 Pruning the client's log

Managing the log at a client is comparatively 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 when connected. An update u can be discarded as soon as it appears in K-stable base version V_i^K, i.e., when the client becomes dependent on the presence of u at a DC.

5. Evaluation

We implement Brie and evaluate it experimentally, in comparison to other protocols. In particular, we show that Brie provides: *(i)* fast response, under 1 ms for both reads and writes to cached objects (§5.3); *(ii)* scalability of throughput with the number of DCs, and small metadata size, linear in the number of DCs (§5.4); *(iii)* fault-tolerance w.r.t. client churn (§5.5) and DC failures (§5.6); and *(iv)* modest staleness cost, under 3% of stale reads (§5.7).

5.1 Implementation and applications

Brie and the benchmark applications are implemented in Java. Brie uses a library of CRDT types, BerkeleyDB for durable storage (turned off in the present experiments), and Kryo for data marshalling. A client cache has a fixed size and uses an LRU eviction policy.

Our client API resembles modern object stores, such as Riak 2.0, Redis, or COPS [2, 25, 31]:

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

Along the lines of previous studies of weak consistency [5, 6, 26, 35], we use two different benchmarks, YCSB and SocialApp, summarized in Table 1.

YCSB [16] 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 closely after WaltSocial [35]. It employs high-level data types such as Sets, for friends and posts, LWW Register for profile information, Counter for counting profile visits, and inter-object references. SocialApp accesses multiple objects in a causal transaction to ensure that operations such as reading a wall page and profile information behave consistently. The SocialApp workload is read-dominated, but the ostensibly read-only operation of visiting a wall actually increments the wall visit counter. The access distribution is uniform.

In order to model the locality behaviour of a client, both YCSB and SocialApp are augmented with a facility to control locality, mimicking social network access patterns [11]. Within a client session, the application draws draws uniformly from a pool of session-specific objects with either 40% (*low locality*) or 80% (*high locality*) probability. Objects not drawn from this local pool are drawn from the global (uniform or zipfian) distribution described above. The size of the pool is smaller than the size of cache.

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, equivalent to a single core 64-bit 2.0 GHz Intel Xeon virtual 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–

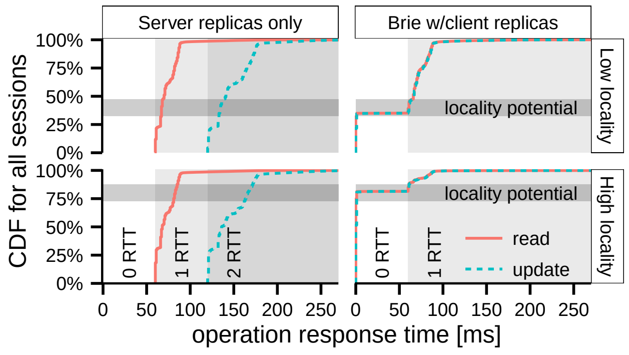


Figure 3. Response time for YCSB operations (workload A, zipfian object popularity) under different system and workload locality configurations.

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 based on the Brie implementation: *(i) Brie mode* (default) with client cache replicas of 256 objects, and refreshed with notifications at a rate ≤ 1 s by default; *(ii) Safe But Fat metadata mode* with cache, but with client-assigned metadata (similarly to PRACTI, or to Depot without cryptography), *(iii) server-side replication mode* without client caches. In this mode, an update incurs two RTTs to a DC, modelling the cost of a synchronous writes to a quorum of servers to ensure fault-tolerance comparable to Brie.

5.3 Response time and throughput

We run several experiments to compare Brie's client-side caching, with server-only geo-replication.

Fig. 3 shows response times for YCSB, comparing server-only (left side) with client replication (right side), under low (top) and high locality (bottom). Recall that in server-only replication, a read incurs a RTT to the DC, whereas an update incurs 2 RTTs. We expect Brie to provide much faster response, at least for cached data. Indeed, the figure shows that a significant fraction of operations respond immediately in Brie mode, and this fraction tracks the locality of the workload (marked "locality potential" on the figure), within a ± 7.5 percentage-point margin, attributable to caching artefacts. The remaining operations require one round-trip to the DC, indicated as 1 RTT. As our measurements for SocialApp show the same message, we do not report them here. These results demonstrate that the consistency guarantees and the rich programming interface of Brie do not affect responsiveness of read and update caching.

In terms of throughput, client-side replication is a mixed blessing: it lets client replicas absorb read requests that would otherwise reach the DC, but also puts extra load of maintaining client replicas on DCs. In another experiment (not plotted), we saturate the system to determine its maximum throughput. Brie's client-side replication consistently improving throughput for high-locality workloads, by 7% up to 128%. It is especially beneficial to read-heavy workloads. In contrast, low-locality workloads show no clear trend; depending on the workload, throughput either increases by up to 38%, or decrease by up to 11% with Brie.

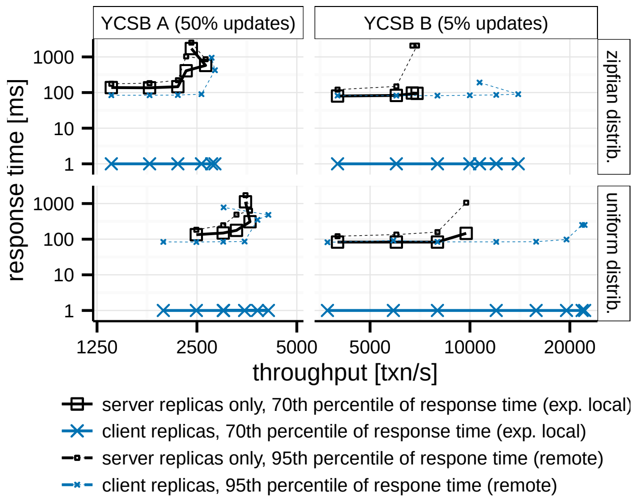


Figure 4. Throughput vs. response time for different system configurations running variants of YCSB.

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, Fig. 4 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 Workload A (up to tens of percent points), but has no effect on the throughput

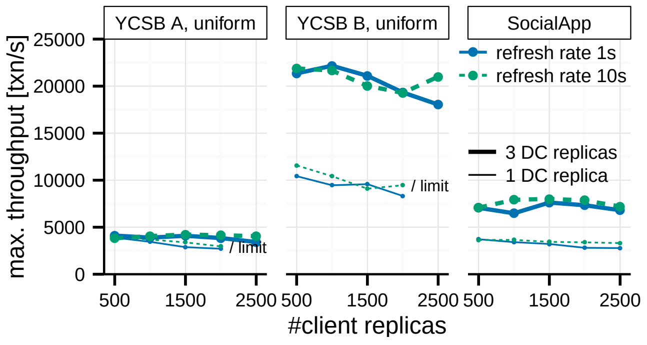


Figure 5. Maximum system throughput for a variable number of client and server (DC) replicas.

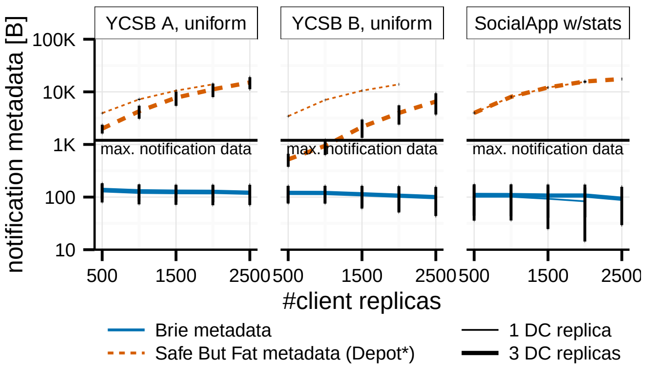


Figure 6. Size of metadata in notification message for a variable number of replicas, mean and standard error. Normalised to a notification of 10 updates.

of read-heavy Workload B. However, we expect the impact of refresh rate to be amplified for workloads with lower rate of notification updates.

5.4 Scalability

Next, we measure how well Brie 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 execute Brie 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 Brie with two different notification rates (every 1 s and every 10 s).

Fig. 5 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.

As to be expected, additional DC replicas increase the system capacity for operations that can be performed at only one replica such as read operations or sending notification messages. Whereas a single Brie DC supports at most 2,000 clients. With three DCs Brie 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 where the cost of maintaining client replicas up to date saturates the DCs, and further clients do not absorb enough reads to overcome these costs. Note that the lower refresh rate can reduce the load at a DC by 5 to 15%.

In the same experiment, Fig. 6 presents the distribution of metadata size 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 Brie "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 [28]. Vertical bars represent standard error. 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 Brie 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. It is independent both from the number of client replicas and from the workload. Increasing the number of DC replicas from one to three causes a negligible increase in metadata size, of under 10 bytes.

In contrast, the classical 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× larger than the actual data this represents a substantial overhead.

5.5 Tolerating client churn

We now turn to fault tolerance. In the next experiment, we evaluate Brie under client churn, by periodically disconnecting client replicas and replacing them with a new set of active clients. At any point in time, there are 500 active clients and a variable number of disconnected clients, up to 5000.

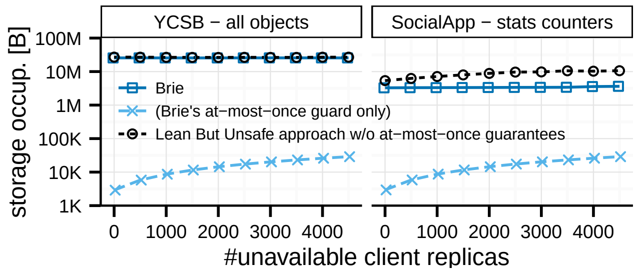


Figure 7. Storage occupation at a single DC in reaction to client churn for Brie and Lean-but-Unsafe alternative.

Fig. 7 illustrates the storage occupation of a DC for representative workloads. We compare Brie's log pruning protocol to a protocol without at-most-once delivery guarantees (Lean But Unsafe).

Brie storage size is approximately constant. 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 can use Lean-but-Unsafe metadata, without Brie'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 idempotent YCSB's LWW-Map, which is naturally idempotent, vs. linear in the number of clients for SocialApp's Counter objects.

5.6 Tolerating DC failures

This experiment studies the behaviour of Brie when a DC disconnects. The scatterplot in Fig. 8 shows the response time of a SocialApp client application as the client switches between DCs. Starting with a cold cache, response times quickly drops to near zero for transactions 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 fail-over succeeds, and the client continues with the new DC. Its response time pattern reflects the higher RTT to the new DC. At 64 s, the client switches back the initial DC, and performance smoothly recovers to the initial pattern.

5.7 Staleness cost

The price to pay for our read-in-the-past approach is an increase in staleness, which our next experiment measures. A read is considered *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. The reason is that the window of vulnerability (the time it takes for an update to become K-stable) is approximately the RTT to the closest DC. For this experiment, we artificially increase the probability of staleness by various means. We run the SocialApp benchmark with 1000 clients in Europe connected to the Ireland DC and replicated in the Oregon DC.

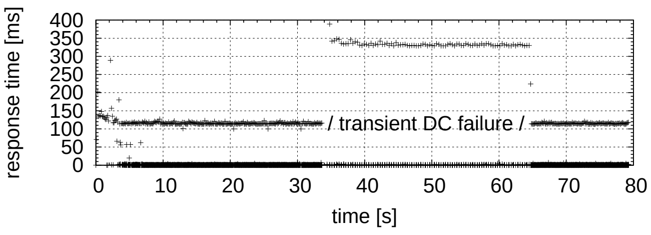


Figure 8. Response time for a client that hands over between DCs during a 30 s failure of a DC.

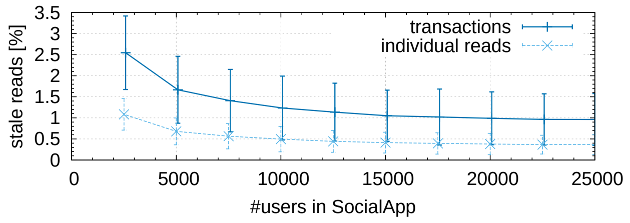


Figure 9. K-stability staleness overhead.

Fig. 9 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 discuss a number of systems that support consistent, available and convergent data access, at different scales. In particular, Table 2 presents the approach to metadata of *causally* consistent systems. Each row groups some systems that share a similar metadata approach. The columns indicate: *(i)* Which replicas assign timestamps; *(ii)* the guaranteed (worst-case) size of metadata summarising a dependency or a version; *(iii)* whether it ensures at-most-once delivery; *(iv)* whether it supports general confluent types.

Client-side replication PRACTI is a seminal work on causal consistency under partial replication [10]. PRACTI uses Safe-but-Fat client-assigned metadata and an ingenious log-exchange protocol that supports an arbitrary communication topology. While such a full generality has advantages, it is not viable for large-scale client-side app deployments backed by the cloud: *(i)* Its fat metadata approach (version vectors sized as the number of clients) is prohibitively expensive (see Fig. 6), and *(ii)* any replica can easily make another unavailable, because of the indirect dependence issue discussed in §3.3.2.

Representative system	Timestamp assignment	Summary metadata max. size, O(#entries)	At-most-once delivery	Support for confluent types
PRACTI [10], Depot [28], COPS [25]	client/any replica	#replicas $\approx$ 1 000 000	yes	weak (COPS) / medium (rest)
Eiger [26], Orbe [19], Bolt-on [6]	DC server (shard)	#servers $\approx$ 100–1 000	no	weak
Walter [35], ChainReaction [5]	DC (full replica)	#DCs $\approx$ 5–10	no	weak
Brie	DC (full replica) client replica	#DCs $\approx$ 5–10 + 1 client entry	no yes	strong

Table 2. Analytical comparison of different classes of metadata used by causally consistent systems.

Our design is strongly inspired by Depot, a (fork-join) causally consistent system that provides a reliable storage on top of untrusted cloud [28]. Depot tolerates Byzantine clients, which our current implementation does not address. Their assumption of Byzantine cloud behaviour requires fat metadata to support direct client-to-client communication. Furthermore, Depot is at at odds with genuine partial replication. It requires every replica to process the metadata of every update, and puts the burden of computing a K-stable version on the client. In the case of extensive DC partitions, it floods all updates to the client. In contrast, Brie relies on DCs to provide K-stable and consistent versions, and uses lean metadata. In the event of failure, Brie provides the flexibility to decrease K dynamically rather than to flood clients.

Both Practi and Depot systems use Safe-but-Fat metadata. They support only LWW Registers, but extension to other confluent types appears feasible.

Recent web and mobile application frameworks, such as TouchDevelop [12], Google Drive Realtime API [14], or Mobius [15] support replication for in-browser or mobile applications. These systems are designed for small objects [14], database that fits on a mobile device [12], or a database of independent objects [15]. It is unknown if/how they support multiple DCs and fault tolerance. This is in contrast with Brie's support for large consistent database, and fault tolerance. TouchDevelop provides a form of object composition, and offers integration with strong consistency [12]. We are looking into ways of adapting similar mechanisms.

Server-side replication A number of geo-replicated systems offer available causally consistent data access inside a DC with excellent scale-out by sharding [5, 6, 19, 25, 26].

Table 2 shows that server-side systems use variety of types of metadata. COPS assigns metadata directly at database clients, and uses explicit dependencies (a graph) [25]. Later publications show that this approach is costly [19, 26]. Consequently, later systems assign metadata at partition replicas [19, 26], or on a designated node in the DC [5, 35]. 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 Brie), but this may limit throughput.

Previous designs are not directly applicable to client-side replication, because: *(i)* their protocols do not tolerate client or server failures; *(ii)* as they assume that data is updated by overwriting, implementing high-level confluent data types is complex and costly (see Fig. 7); *(iii)* the size of their metadata can grow uncontrollably.

Du et al. [20] make use of full stability, a special kase of K-stability, to remove the need for dependency metadata in messages, thereby improving throughput.

Integration with strong consistency Some operations or objects of application may require stronger consistency, which requires synchronous protocols [21]. 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 [24, 35]. We speculate that these techniques are applicable to Brie, grounded on preliminary experience.

Theoretical limits Mahajan et al. [27] prove that causal consistency is the strongest achievable model in an available, convergent, *full* replication system. We conjecture that these properties are not simultaneously achievable under partial replication, and demonstrate how to weaken one of the liveness properties. Bailis et al. [7] give an argument for a similar result for a client switching server replicas, but do not take into account the capabilities of a client replica.

7. Conclusion

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

Our experiments confirm that Brie is able to provide immediate and consistent response on reads and updates on local objects, and maintain the throughput of a server-side replication system, or better. The novel form of metadata allows the system to scale to thousands of clients with constant size objects and metadata, independent of the number of available and unavailable clients. Our fault-tolerant protocols handle failures nearly transparently.

Many of these properties are due to a common principle demonstrated by Brie design: client buffering and controlled staleness can absorb the cost of scalability, availability, and consistency. Staleness cost is moderate and well separated.

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 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, 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] J. Du, C. Iorgulescu, A. Roy, et al. Closing the performance gap between causal consistency and eventual consistency. In *Workshop on Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, Netherland, 2014.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

C Antidote API

This section describes the API for the different layers in Antidote. The notation here follows the implementation in Erlang. Every partition of the datastore is managed by its own vnode.

The types appearing in the signature are defined as follows:

- A log node *lognode() :: {Partition::integer(), node()}* represents a process (virtual node/vnode) in the local Erlang VM managing the log for a partition of the data store.
- The *log() :: [Partition::integer()]* consists of a list of partitions.
- An operation *op() :: #operation{operation_ number=op_ id(), payload=term()}* consists of an operation identifier and (optionally) the payload.
- The operation id *op_ id() :: {Clock::non_ neg_ integer(), node()}* uniquely identifies an operation using a per-node clock time stamp.
- The key *key() :: term()* uniquely identifies a CRDT object in the key-value data store.
- The type *type() :: atom()* denotes the CRDT type for a CRDT object.

C.1 Logging Layer

Log

- *append(Node::lognode(), Log::log(), Record::term())*
  Appends *Record* to the persistant log.

  Input:

  - *Log* is the identifier for the log to which the record belongs to.
  - *Node* is the virtual node which hosts the *Log*.
  - *Record* is any operation defined by the transaction layer to be appended to the log.

  Return:

  - *{ok, {Node, op_ id}}* if the append was successfull. *op_ id* is the unique identifier for the newly appended record.

- *read(Node::lognode(), Log::log())*
  Returns all records stored in the log identified by *Log*

  Input:

  - *Log* is the identifier for the log.
  - *Node* is the virtual node which hosts the *Log*.

  Return:

  - *{ok, {Node, Ops::op()}}* if the read was successful. *Ops* is a list of records with their op_ids.

- *read_from(Node::lognode(), Log::log(), From::op_id())*
  Returns all records stored in the log identified by *Log* that were inserted after the record with id *op_id*.

  Input:

  - *Log* is the identifier of the log.
  - *Node* is the virtual node which hosts the *Log*.
  - *op_id* is the operation identifier

  Return:

  - *{ok, {Node, Ops}}* if the read was successful. *Ops* is a list of records with their op_ids.

Materializer

- *read(Key::key(), Type::type(), SnapshotTime::vectorclock())*
  Reads a particular version of CRDT identified by Key from the in-memory cache.

  Input:

  - *Key* : The key to be read
  - *Type* : The type of key
  - *SnapshotTime* : The version to be read. It is a vectorclock defined by the transactional causal+ protocol.

  Return:

  - *{ok, CRDT::term()}* : The state of the Key as defined by the CRDT type

- *update(Key::key(), Operation::op())*
  Appends an operation for a Key to the in-memory cache of the materializer. A successfull append returns *ok* which guarantees that further read operations include this operation if the version requested contains this operation.

C.2 Transactions

The transactional interface of Antidote supports three types of transactions. The clients can use these interface to execute operations on CRDTs in Antidote.

Single Key operations

- $append(Key :: key(), Type :: type(), Operation :: term())$ This executes the append operation on Key as if it is a transaction with single update operation.

  Input:

  - *Key* : The key to which the operation is applied.
  - *Type* : The type of the key
  - *Operation* : A valid operation on the type of the key

  Returns:

  - *ok* if append was successfull
  - *{error, Reason}* if there is any error

- $read(Key :: key(), Type :: type)$ This reads the latest value of Key available in the data centre.

  Input:

  - *Key* : The key to which the operation is applied.
  - *Type* : The type of the key

  Retuns:

  - *{ok, Val}* - if the read was successful. *Val* is the value of the key which is read
  - *{error, Reason}* - if there was any error

- $clocksi\_read(ClientClock :: vectorclock(), Key :: key(), Type :: type())$ Returns the latest value of Key available in the data centre whose version is atleast *ClientClock*. If the data centre does not have that version, it waits until the version is available.

  Input:

  - *ClientClock*: the last clock the client has seen from a successful transaction.
  - *Key*: the key intended to be read.
  - *Type*: the CRDT type, a valid type from $riak\_dt$.

  Returns:

  - *{ok, ReadResult::term()}* an ok message along with the result of the read operation.
  - *{error, Reason::term()}* an error message in case of a failure.

Multiple Key updates

- $clocksi\_bulk\_update(ClientClock :: vectorclock(), Operations :: [op()])$

  The client can execute multiple updates atomically with the transactional causal+ semantics.

  Input:

  – *ClientClock* : the last clock the client has seen from a successful transaction. It can be empty if the client doesnot require to see its own writes from previous transactions.
  – *Operations* : the list of the update operations in the transaction.

  Returns:

  – *{ok, CommitTime}* : an ok message along with the transaction's commit time. This CommitTime can be used by the client in following transactions to guarantee "read your own writes" semantics.
  – *{error, Reason::term()}* :an error message in case of a failure.

Interactive transactions

- $clocksi\_istart\_tx(ClientClock :: vectorclock())$ Starts a new interactive transaction.

  Input:

  – *ClientClock* : the last clock the client has seen from a successful transaction.

  Returns:

  – *{ok, TxId::tx_id()}*: an ok message along with the transaction Id that will be used by the client every time an operation for that transaction is sent. This snapshot time will be used for defining the snapshot the transaction reads from.
  – *{error, Reason::term()}*: an error message in case of a failure.

- $clocksi\_iread(TxId :: tx\_id(), Key :: key(), Type :: atom())$

  Reads a key within an interactive transaction according to the transactional causal+ semantics.

  Input:

  – *TxId*: the transaction ID, as returned by the start transaction function.
  – *Key*: the key intended to be read.
  – *Type*: the CRDT type, a valid type from $riak\_dt$.

  Returns:

  – *{ok, ReadResult}*: in case of success.

  - *{error, Reason}* :error message in case of a failure.

- $clocksi\_iupdate(TxId :: tx\_id(), Key :: key(), Op : term())$

  update a key within an interactive transaction according to the transactional causal+ semantics. The update is not visible until the transaction commits.

  Input:

  - *TxId*: the transaction ID, as returned by the start transaction function.
  - *Key*: the key that the operation updates.
  - *Op*: the operation and parameters that the update operation receives, as in the *riak_dt* implementation of the CRDT type that is updated.

  Returns:

  - *ok* when the updates successes.
  - *{error, Reasonn}*: error message in case of a failure.

- $clocksi\_iprepare(TxId :: tx\_id())$ Sends the prepare command for starting to commit the transaction.

  Input:

  - *TxId* : the transaction ID, as returned by the start transaction function.

  Returns:

  - *{ok, PrepareTime::integer}*, where PrepareTime is the time that the transaction will commit with.
  - *{error, Reason::term()}*: error message in case of a failure.

- $clocksi\_icommit(TxId :: tx\_id())$ Sends the commit command for the transaction identified by TxId.

  Input:

  - *TxId*: the transaction ID, as returned by the start transaction function.

  Returns:

  - *{ok, CommitTime::vectorclock()}*, the time that is used in the transaction's commit record.
  - *{error, Reason::term()}*: error message in case of a failure.