Partial Replication of Conflict-Free Replicated Data Types

Hugo Guerreiro
hugo.guerreiro@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisors: Professor Luís Rodrigues and Professor Nuno Preguiça)

Abstract. Conflict-free replicated data types (CRDT) are a class of data structures designed to simplify the operation of distributed systems that require data to be replicated in different nodes. CRDTs are designed in such a way that different replicas can execute concurrent operations in different orders and still converge to the same consistent state when the system becomes quiescent. In this way, CRDTs avoid the coordination costs required for preventing conflicts from occurring (such as executing operations in total order) or complex conflict resolution procedures. CRDTs are now well understood in the case of full replication, i.e., when all replicas maintain the entire state of the CRDT. However, with CRDTs that store large quantities of data, full replication may be infeasible; one may be forced to shard the CRDT state and let different replicas replicate different shards, i.e., to support partially replicated CRDTs. While classical CRDT algorithms will ensure the consistency of each shard, it may be difficult to ensure that clients observe a consistent state of the CRDT when they access multiple shards. Also, the data access patterns may require the system to change which nodes replica each shard dynamically.
Additionally, in some cases, static partitioning of the CRDT may not be feasible, and it may be useful to dynamically re-allocate objects to shards. In this work, we study ways to express different types of partially-replicated CRDTs, including CRDTS where the number of shards, the assignment of objects to shards, or the number of replicas of each shard may change in runtime. We also propose algorithms to ensure that the use of partial-replication does not affect the consistency of the CRDT.

Key words: CRDT · Partial replication · Causal consistency

Table of Contents

1 Introduction 3
2 Goals 4
3 Causality and Causal Consistency 4
  3.1 Logical Clocks 5
  3.2 Vector Clocks 5
  3.3 Causal Consistency 6
4 Conflict-free Replicated Data Types 6
  4.1 System Model 7
  4.2 Synchronization Model 7
    4.2.1 Operation-Based Replication 7
    4.2.2 State-Based Replication 8
    4.2.3 Delta-CRDT Replication 10
  4.3 Basic CRDTs 10
    4.3.1 Counters 11
    4.3.2 Registers 13
    4.3.3 Sets 14
    4.3.4 Other CRDTs 15
5 Related Work 15
  5.1 Partial Replication of CRDTs 16
    5.1.1 Conflict-free Partially Replicated Data Types 16
    5.1.2 Composing Partial CRDT Replicas 18
    5.1.3 Non-Uniform Replication 19
  5.2 Causally Consistent Partially Replicated Systems 21
    5.2.1 Causal Consistency vs Partial Replication 21
    5.2.2 An Overview of Current Solutions 22
6 Architecture 25
  6.1 System Model 26
7 Evaluation 28
8 Scheduling of Future Work 28
9 Conclusions 29

1 Introduction

The use of data replication is almost unavoidable in modern distributed systems. First, data replication is a fundamental technique to provide fault-tolerance, a key requirement in most systems. Second, when the applications have clients that reach the system from different geographic locations, only replication can ensure that these clients access data with low latency. Unfortunately, with data replication, comes the risk that clients may observe an inconsistent view of said data. The problem of data consistency in systems that maintain replicated data has been widely studied [1]. In this context, several consistency models have been proposed that make different tradeoffs between the guarantees provided to the applications and the costs involved in ensuring those guarantees. Some consistency models hide from the application the fact that data is replicated, such that application designers can reason about the system as if the data is stored centrally in a single node. These models are often said to offer *strong consistency*; relevant examples are linearizability [2] and serializability [3].

Unfortunately, it has been shown that strong consistency cannot be enforced without tight coordination among replicas, in particular without running consensus [4, 5]. In turn, consensus cannot be solved in a pure asynchronous system and, in systems that can be augmented with a failure detector, can block in the face of unfavorable network conditions. This leads to the observation that it is impossible to offer consistency, availability, and partitioning tolerance [6] simultaneously. Weaker consistency models, such as causal consistency and eventual consistency [7], have also been proposed. These models can be implemented without executing consensus and can offer better availability. Typically, such models work by allowing any replica to accept updates and subsequently propagate them to other peers in the background. However, these models allow for concurrent operations to be accepted at different replicas, sometimes generating conflicts that need to be solved by complex conflict resolution procedures. Developing these mechanisms is typically an ad hoc and error-prone process.

Conflict-free replicated data types (CRDT) are a class of data structures designed to simplify the operation of distributed systems that require data to be replicated in different nodes. CRDTs are designed in such a way that different replicas can execute concurrent operations in different orders and still converge to the same consistent state when the system becomes quiescent. This type of quiescent consistency [8, 9] named strong eventual consistency [10](SEC) is more strict than weak consistency and gives to applications additional safety guarantees.

CRDTs are now well understood in the case of full replication, i.e., when all replicas maintain the entire state of the CRDT. However, with CRDTs that store large quantities of data, full replication may be infeasible; one may be forced to shard the CRDT state and let different replicas replicate different shards, i.e., to support partially replicated CRDTs. While classical CRDT algorithms will ensure the consistency in each shard, it may be challenging to ensure that the client observes a consistent state of the CRDT when they access multiple shards. Also, the data access patterns may require the system to change which nodes

replica each shard dynamically. Additionally, in some cases, static partitioning of the CRDT may not be feasible, and it may be useful to dynamically reallocate objects to shards. In this work, we study ways to express different types of partially-replicated CRDTs, including CRDTS where the number of shards, the assignment of objects to shards, or the number of replicas of each shard may change in runtime. We also propose algorithms to ensure that the use of partial-replication does not affect the consistency of the CRDT.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 5, we present all the background related to our work. Section 6 describes the proposed architecture to be implemented and Section 7 describes how we plan to evaluate our results. Finally, Section 8 presents the schedule of future work and Section 9 concludes the report.

2 Goals

This work addresses the problem of partial replication in the context of CRDTs. More precisely:

> *Goals:* We aim at designing a common framework to express different types of partially-replicated CRDTs, including CRDTs where the number of shards, the assignment of objects to shards, or the number of replicas of each shard may change in runtime. We also aim at designing, implementing, and evaluating algorithms to enforce the consistency of partially-replicated CRDT.

To achieve this goal, we plan to depart from a novel system model that allows any type of CRDT to be partially replicated. We will augment this system to encompass dynamic assignments of data to shards and also of shards to replicas. We will try to make our system powerful enough to support other forms of partially-replicated CRDTs such as non-uniform replication [11] and conflict-free partially replicated data types [12]. At the implementation level, we expect to develop algorithms that will allow for more general synchronization patterns than the centralized model described in [13].

The project will produce the following expected results.

> *Expected results:* The work will produce i) a specification of partially replicated CRDTs; ii) a prototype that shows the feasibility of implementing the proposed specification, iii) an extensive experimental evaluation that compares the performance and flexibility of the proposed system with previous systems that provide different forms of partially-replicated CRDTs.

3 Causality and Causal Consistency

Time is an essential concept when reasoning about how events are ordered in a system. Intuitively, every event happens at a certain point in time, and

using this physical time to order such events would be the ideal way to do so. Unfortunately, since perfectly synchronizing clocks in a distributed system is unfeasible [14], other methods had to be sought out.

One way to order the events in a system is through their causal relationships. Lamport [14] generalized this concept through the *happened-before* relation ($\rightsquigarrow$) of two events. An event e_1 is said to have *happened-before* an event e_2 ($e_1 \rightsquigarrow e_2$) iff:

(i) If e_1 and e_2 are events that happened in the same process and e_1 occurred before e_2 then $e_1 \rightsquigarrow e_2$
(ii) In the case of a message being sent; if e_1 is the event responsible for sending said message, and e_2 the event reading it, then $e_1 \rightsquigarrow e_2$
(iii) If $e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$ then $e_1 \rightsquigarrow e_3$

3.1 Logical Clocks

Logical clocks were introduced alongside the definition of happens-before to capture these relationships between events. Logical clocks are, in their essence, counters that increase monotonically and do not need to share any dependency with real physical clocks. A logical clock L_i associated with an arbitrary process p_i is defined as follows:

(i) Before any event happens at p_i, L_i is incremented: $L_i := L_i + 1$
(ii) Everytime process p_i sends a message, L_i is incremented and its value is L_i is *piggybacked* in the message.
(iii) On the receiving site, process p_j computes $L_j := max(L_j + 1, L_i)$

The use of logical clocks is sufficient to enforce causality. In fact, given two events e_1 and e_2 we have $e_1 \rightsquigarrow e_2 \Rightarrow L(e_1) < L(e_2)$, the contrary, however, is not true ($L(e_1) < L(e_2) \nRightarrow e_1 \rightsquigarrow e_2$).

3.2 Vector Clocks

To overcome the aforementioned limitation, vector clocks were introduced [15, 16]. A vector clock joins multiple logical clocks, one for each process in the system - e.g., if the system has N processes, the vector clock will contain N logical clocks. Similarly to logical clocks, each process p_i keeps a vector clock V_i which is updated every time an event happens, and the vector clock is piggybacked in every message sent by a process. The rules for updating a vector clock are:

(i) Each entry in the vector V_i is initialized to 0
(ii) Each time an event occurs at process p_i , its vector clock is incremented $V_i[i] = V_i[i] + 1$
(iii) When process p_i receives a message from another process p_j, the value of the vector clock is set to the pairwise maximum of each entry in both clocks: $V_i[k] := max(V_i[k], V_j[k])$, for $k = 1, 2, ..., N$.

Unlike logical clocks, this mechanism ensures that for two events e_1 and e_2 the following is true: $e_1 \rightsquigarrow e_2 \Rightarrow L(e_1) < L(e_2)$; $L(e_1) < L(e_2) \Rightarrow e_1 \rightsquigarrow e_2$. This property makes it possible to compare two vectors in such a way that we can not only identify whether one event happened before other but also if two events are concurrent.

Although vector clocks are a better approximation of causality than logical clocks, they have the disadvantage of growing linearly in size with the number N of processes in the system. As N grows large, so does the penalty incurred in terms of bandwidth and storage. Several other causality tracking mechanisms and optimizations have been proposed in the literature [17–24].

3.3 Causal Consistency

The happens-before relation was initially conceived in the context of message-passing systems but it is also relevant for systems based on shared memory. Causal consistency [7] is a consistency model for shared memory systems that ensures that reads always return values that are consistent with the happens-before relation among read and write operations, i.e., each process observes a state that results from an execution of write operations that complies with causal ordering (concurrent updates, may be applied in any order). This means that once the effect of a certain write operation becomes visible at a given process, so do the effects of all write operation in its causal past.

This consistency model is really important in distributed systems. Reasoning about causality bonds naturally with our notions of how time behaves; it provides better guarantees than eventual consistency while still tolerating partitions and has been proven to be the strongest consistency model that is also available under partition [4, 5].

Notice that by allowing concurrent updates to execute at different orders in different processes, the states may diverge. A slightly stronger model named *Causal+* [25] has been proposed, where the additional condition of eventual and independent conflict resolution of concurrent updates needs to be ensured. This condition is such that the states of every process will eventually reach the same value. Conflict-free replicated data types (CRDTs), which will be presented in the next section, are a known way to implement causal+ consistency.

4 Conflict-free Replicated Data Types

Commutative or convergent replicated data types (CRDT) are a class of distributed data structures that can be replicated throughout different nodes and have two important properties:

(i) Updates to a certain replica can be done without any coordination and conflicts are automatically resolved
(ii) Replicas are guaranteed to eventually and deterministically reach the same state given that they saw the same set of updates

The asynchronous nature of CRDTs makes them useful for applications that require low update latencies, but that can trade-off for less data consistency. In the following chapter, we take a look at the CRDT's system and synchronization model and conditions for convergence as well as some known basic CRDTs.

4.1 System Model

The distributed system considered for CRDTs is composed of nodes that communicate over an asynchronous network that can partition and recover. In the case of partitioning, nodes can continue to work without communicating with other nodes.

Processes may crash due to a non-byzantine fault. They may recover along with the memory that is also recovered to the state previous to the crash.

Processes hold CRDT objects. An object is a collection of simple (base) data types (e.g. Integers, Strings, sets, among others) and other objects that are stored under its payload (or content). CRDT objects expose certain operations in order to manipulate said payload.

4.2 Synchronization Model

Replicas of commutative replicated data types (CmRDT), and convergent replicated data types (CvRDT) are guaranteed to deterministically converge to the same state given that all updates eventually reach all replicas. Guaranteeing that all changes propagate to every replica is part of what is called the synchronization model, and two types are considered for CRDTs, namely state and operation based, which are inherently tied with CmRDTs and CvRDTs respectively. In the following section, we take a look at how these data types achieve convergence, the necessary conditions for it to happen as well as their synchronization models.

4.2.1 Operation-Based Replication

In the operation-based synchronization model, replicas execute operations locally at the replica chose by the client (called the source replica) and afterward propagate them to all other replicas. As described, operations are executed in two stages:

(i) **Prepare phase (or atSource phase) -** The update operation is prepared for execution at the source replica. No side-effects happen;
(ii) **Downstream phase (or effect phase) -** The operation is executed locally and asynchronously transmitted to all the other replicas.

Queries can be executed entirely at a single replica without the need to propagate to other replicas; however, since updates change the state, the same cannot be done. To this end, this synchronization model assumes that updates are delivered according to a causally consistent order with the caveat that all

concurrent operations must commute, i.e., the order in which operations are applied does not matter and always result in the same final value. This requirement is achieved by using CmRDTs (described in Section 4.2.1.1) as the conciliation medium for operations that are applied at a specific replica.

Apart from the causally consistent delivery order, replicas must also only see each update exactly once; therefore, an underlying system that provides reliable broadcast is assumed to exist.

4.2.1.1 ***Commutative Replicated Data Types (CmRDT)*** Commutative replicated data types have one crucial characteristic: concurrent update operations commute.

As previously described, this property is mandatory for the correctness of the synchronization model in the sense that the result of two concurrent operations is the same independently of the order in which they were executed. Furthermore, this reduces the need for additional consensus related communications.

With respect to convergence, if a delivery order exists for the updates that were made and given that all concurrent operations commute, then all replicas converge to the same state. A delivery order cannot be stricter than causal delivery, which is sufficient for all CmRDTs. Nevertheless, weaker orderings can be employed for some data types.

The synergy between operation based synchronization and the convergence of CmRDTs erupts from the fact that any two replicas of a CmRDT eventually converge under reliable broadcast channels that deliver operations in the specified delivery order. The proof for this result can be consulted in detail in [10,26].

Correctly designing a CmRDT can be narrowed down to proving two points:

- Showing that a delivery order exists for all updates;
- Proving that concurrent updates commute.

4.2.2 State-Based Replication

In the state-based replication model, updates are executed locally at the replica upon which the operation was invoked (source replica). Afterward, the entire modified state is propagated to other replicas that will apply a merge function between their local state and the received payload. This merge function results in a new state that includes all the updates that were executed at the source replica. CvRDTs are a theoretically sound approach to design such merge functions. CvRDTs are further discussed in Section 4.2.2.1.

It might be useful not to execute a certain operation if a condition is not met. For that end, one can specify pre-conditions that are evaluated before executing a particular operation. If an operation meets all the pre-conditions, then it can be enabled for execution.

As long as replicas are constantly transmitting their state to their peers and the replica communication network forms a connected graph, all updates are eventually seen by every replica.

4.2.2.1 ***Convergent replicated data types (CvRDT)*** Just like its counterpart CmRDT, Convergent replicated data types also have a payload, and their replicas are able to converge to the same state deterministically. CvRTs are equipped with a merge(x, y) function that takes two states x,y, and joins them in a conflict-free manner.

More formally, CvRDTs lay its foundations in order theory and the notion of semilattice. CvRDTs have three main properties:

(i) The values from its payload are taken in a semilattice;
(ii) The $merge(x, y) = x \sqcup y$ function converges towards the least upper bound (LUB) of the two states;
(iii) When merging, the state of the object increases in a monotonic fashion, i.e., it never decreases in size.

The combination of a monotonic increasing state that takes its values in a semilattice is called a monotonic semilattice. These properties combined with state-based synchronization are sufficient conditions of convergence for state-based CRDTs [10, 26]

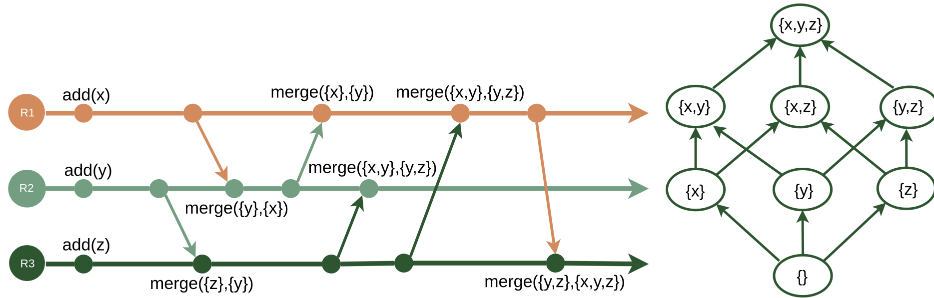

Fig. 1: State evolution of three replicas of a state-based CRDT

Intuitively, the evolution of the state of a CvRDT can be explained with a simple Hasse diagram, such as the one in Figure 1. Additionally, in Figure 1 we have a possible serialization of the events in the system that will lead to a coherent final state between all replicas according to the state evolution in the respective Hasse diagram. In these two diagrams, we consider the payload of the object to be a set where x, y and z are atoms of said set. For the example's sake, we can consider that for two sets S and T we have $S \leq T \Leftrightarrow S \subseteq T$; this defines a partial order. Furthermore, if we specify the merge function as $merge(S, T) = S \cup T$ we are computing the least upper bound of the two sets. By joining these two definitions, we have created a join-semilattice. Assuming objects are not removed from these sets, we have a set that only increases in size, therefore it is monotonic. Altogether, we have a monotonic semilattice, and this

object can be called a convergent replicated data type. The exposed example is, in reality, an actual state-based CRDT called Grow only set or G-Set.

4.2.3 Delta-CRDT Replication

Both state and operation based CRDTs have their disadvantages. On the one side, state-based CRDTs require the underlying system to propagate the entire state when updates are made; This inevitably impacts the performance of the system as the size of the CRDT object increases. On the flip side, when multiple updates are made to the same object, propagating all the operations might be inefficient as opposed to sending the state once. As an example, consider a replicated counter object that can be incremented or decremented; propagating all the updates made to this object is heavier to the system than propagating once, what has changed since the two replicas last communicated.

This trade-off between state-based and operation-based CRDTs is studied by delta-state CRDTs [27], which combine features from the two other types of CRDTs. Updates to Delta-state CRDTs generate deltas which can be thought of as the difference of the state before and after the update was executed. The effect of one or more updates is encoded as delta-mutators, which are then propagated between replicas. Delta-mutators act like updates as in operation-based CRDTs but have the mergeable characteristic of state-based CRDTs, essentially making this type of CRDTs a more practical version of state-based CRDTs.

Surprisingly, using δ-CRDTs does not necessarily improve the amount of redundant state propagated between replicas [28]. Additionally, the initial communication for δ-CRDT replicas is heavy and requires the entire state to be propagated between the two replicas [29]. Addressing both these problems can be done using similar approaches that, in their essence, consist in the computation of the optimal delta-mutators to be propagated [28, 29].

4.3 Basic CRDTs

In this section, we take a look into some existing CRDTs that are fairly simple in complexity. An issue that comes packaged with designing CRDTs is the fact that different objects might have multiple possible conflict resolution semantics. As an example, consider trying to concurrently add and remove the same element from a shared set. Noticeably, there is not a single possible outcome from these operations: the result can be either the object being added to the set or not. From an application point of view, different conflict resolution semantics may make sense depending on the type of application. To this end, CRDT specifications decide the semantics of the object, and the application developer decides if that semantic makes sense for the desired use case. A large number of CRDT specifications have been proposed [10, 26, 27, 30, 31]. Next, we present only a few examples.

4.3.1 Counters

A counter is a data type that has an integer as its payload and supports operations to either increment or decrement said value. Since these updates are naturally commutative, it is expected for the CRDT to converge to the sum of the increments minus the sum of decrements. Albeit only having a single concurrency semantics, some different constructs are still considered.

4.3.1.1 ***Operation-based counter*** The operation-based counter is one of the most intuitive CRDTs. Since addition and subtraction commute, concurrent operations may be applied in any order; in fact, given that this CRDT is always incremented or decremented by the same amount, all operations may be applied in any order.

Algorithm 1 Specification of an Operation-based counter. Specification taken from [26]

```
 1: payload integer i
 2:     initial 0
 3: query value() : integer j
 4:     let j = i
 5: update increment()
 6:     atSource()                      ▷ Empty: no prepare phase is needed
 7:     downstream()                    ▷ No precondition: delivery order is empty
 8:         i := i + 1
 9: update decrement()
10:     atSource()                      ▷ Empty: no prepare phase is needed
11:     downstream()                    ▷ No precondition: delivery order is empty
12:         i := i − 1
```

The specification for this CRDT is presented in Algorithm 1. To be as illustrative as possible of how operation-based CRDTs are designed, we will go in detail through this specification. In line 1, we can see the specification of the payload of the object (an integer with an initial value of 0 in this case). This *payload* is instantiated at all replicas. Operation-based CRDTs typically support two types of operations: *queries* and *updates*. Queries execute locally at each replica. In line 3 is defined a query operation that simply *returns* the current value of the counter. While this operation is rather simple, some complex operations require certain conditions to be met before being allowed to execute. These conditions are named *preconditions* and are placed at the beginning of the operation definition.

As previously explained in Section 4.2.1, update operations are executed in two steps: *atSource* and *downstream*. In lines 5 and 9, we can see the definitions of the *increment* and *decrement* update operation, respectively. In this CRDT, only the downstream phase needs to be specified, and no precondition is needed since updates may be executed in any order. The downstream phase

increments/decrements the counter locally and then asynchronously propagates the operation to the other replicas.

4.3.1.2 ***State-based increment only counter (G-counter):*** This CRDT works by maintaining a vector where each entry is the value for the counter at that replica. The value of the object is the sum of all entries in the vector. When merging, we take the max between each position in the payload vector and the receiving vector.

Similarly, as we did for the operation-based counter in Section 4.3.1.1, let us take a glance over the specification for this CRDT. Specifications for state-based CRDTs differ from the ones for operation-based in some aspects: (i) update operations do not require to be executed in two steps and can be immediately applied to the local state; (ii) two additional methods need to be specified, namely *compare* and *merge*.

Algorithm 2 Specification of a state-based increment only counter. Specification taken from [26]

1: **payload** $integer[n]\ P$ ▷ One entry per replica
2: **initial** $[0, 0, ..., 0]$
3: **update** $increment()$
4: **let** $g = myID()$ ▷ g: source replica
5: $P[g] := P[g] + 1$
6: **query** $value() : integer\ v$
7: **let** $v = \sum_i P[i]$
8: **compare**$(X, Y) : boolean\ b$
9: **let** $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
10: **merge**$(X, Y) : payload\ Z$
11: **let** $\forall i \in [0, n-1] : Z.P[i] = max(X.P[i], Y.P[i])$

Similarly to Specification 1, in line 1, we define the payload for this object. In line 3, we define the increment operation that will increment the position in the vector corresponding to the local replica. In line 6, we define the *query* operation, which returns the sum of all the counters in the vector. Operations *compare* and *merge* are both defined in lines 8 and 10, respectively. The *compare(X, Y)* receives two states and is responsible for evaluating if there exists a *partial order*[1] between these two states. In this particular CRDT, this relation is defined by applying a $\leq$ comparison over each entry in one vector with the corresponding entry in the other vector. This operation is essential for ensuring that the state increases in a monotonic way.

[1] In more simple terms, one can think of this as a comparison to check if one abstract state is greater than the other. Although not mathematically correct, this intuition holds for most practical scenarios.

The other operation is the merge(X,Y) that, given two states, computes the LUB between them. For the increment-only counter, this is done by merely taking the max between each corresponding entry in the two vectors. By doing so, we guarantee that the newly generated vector will respect the compare(X,Y) function when applied together with the two input vectors of the merge function.

4.3.1.3 ***State-based PN Counter:*** The PN-counter follows the same logic as the G-Counter but supports decrement operations. To allow this additional behavior, a second dedicated vector is stored in each replica to represent the decrements. The value of the CRDT is the sum of the increment vector minus the sum of the decrement vector.

4.3.1.4 ***Non negative Counter:*** Allowing for non-negative counters while not breaking the semantics of the data structure (the value of the object converges towards the global number of increments minus the number of decrements) proves to be quite challenging. One possible solution is not to decrement more times than there have been increments.

4.3.2 Registers

Registers can be seen as a variable to which we can assign a certain opaque value. Since updates to registers behave as separate events and have no relation between them, sequential semantics must be preserved. To this end, two concurrency semantics have been proposed:

4.3.2.1 ***Last Writer Wins Register:*** The last writer wins register maintains an arbitrary object as its payload. An order of the update operations is formed by associating a unique, totally ordered, and causally consistent timestamp to every event. In general, when faced with two concurrent writes, the LWW register chooses the one which was written last.

In the state-based version of this CRDT, the merge-operation accepts the value from the replica whose timestamp is the most recent.

On the operation-based version, the update operation is executed immediately locally. In the *Downstream* phase, the operation is only accepted if the timestamp is greater than the one stored at the receiving replica. Concurrent updates are guaranteed to commute given the fact that each timestamp is unique and totally ordered.

4.3.2.2 ***Multi Value Register (MV-register):*** In the LWW register, some of the values that were written concurrently might be discarded. For some applications, this semantic might not make sense, and the application developer may need to have access to all values that reached the system at a given time. In the state-based Multi-value register CRDT, all the values that were written in a concurrent fashion are kept. To detect concurrently written values, the data structure maintains pairs of $< Value, Versionvector >$. When a merge is required, the replica only accepts the pairs that do not dominate one over the

other. In the case of concurrent writes, since no update is the dominant one, both values are kept.

4.3.3 Sets

A Set is a well-known mathematical data structure that has many useful applications in various fields of research. It is one of the core basic data structures that lay the foundations for more complex data types like Containers, maps, and graphs. In the case of CRDTs, two operations are considered in sets: $add(e)$ that allows an element to be added (equivalent to a set *union*) and $rmv(e)$ that removes a certain element (equivalent to a set *minus*). Although simple, these operations do not commute, and as a result of that, a CRDT can only approximate the sequential specifications of a set [26]. Multiple concurrency semantics have been proposed for elements that are added and removed concurrently.

4.3.3.1 ***Grow-only set (G-set):*** The grow-only set is a simple yet useful CRDT. It maintains a set as its payload and only allows update and lookup operations to be executed. In Section 4.2.2, an intuition was already brought about the state-based version of a G-Set. Regarding the operation-based version, since the *union* operator of a set is commutative, concurrent adds can be executed in any order. Given that all considered operations commute, this data structure is a CmRDT and under operation-based synchronization, it is a CRDT.

4.3.3.2 ***Two phase set (2P- set) and U-set:*** The two-phase set allows both adding and removing operations with the caveat that once an element has been added and removed, it cannot be added again. The state-based version of this CRDT works by combining two G-set CRDTs, one for adding new elements and another for tracking elements that were removed. This second CRDT is known as the *tombstone*.

Intuitively, this data-structure is a CvRDT since it is the combination of two already proven to be CRDTs. Regarding the operation-based version, it is given that concurrent operations of the same type ($add(e) \| add(e)$, $rmv(e) \| rmv(e)$) commute and since a pre-condition ensures that a remove of an element can only be executed if the corresponding *add* has been executed, concurrent updates of the type $add(e) \| rmv(e)$ cannot happen therefore this data structure is a CmRDT.

Additionally, the 2P-set can be simplified under the assumption that all elements are unique, and as a consequence of that, a removed element will never be added again. If additionally, a downstream pre-condition ensures that the operation add(e) is delivered before remove(e), there is no need to keep a tombstone set. This construct is named U-set.

4.3.3.3 ***LWW-element-Set:*** Another concurrency semantic is the one considered by the last writer wins set. In this CRDT, a total order is defined among all updates, and in the presence of concurrency issues, the update that was written last according to the specified order is the one that prevails. Intuitively, when

faced with a concurrent add and remove of some element e, if the *add* update was executed after the remove operation, the element will be in the set after the merge.

4.3.3.4 ***Observed remove set (OR-set):*** The concurrency semantics of the Observed remove set (also known as add-wins set) is quite intuitive. In this CRDT, both $add(e)$ and $rmv(e)$ operations are supported. In the case of concurrent *add* and *rmv* updates, priority is given to the *add* operation, which makes the element be considered in the final state. A variation of this semantics is the **remove-wins set**, which contrary to its counterpart OR-set, gives priority to remove operations.

4.3.4 Other CRDTs

In the literature, more complex constructs have been formulated. Among others, more advanced CRDTs include: Lists [32], Maps [33], Graphs [34] and JSON objects [31].

5 Related Work

Thus far, CRDTs work under the assumption that the state of an object is kept in its entirety on each replica. In practical scenarios where the devices that hold the objects are limited in memory, transferring large amounts of data is suboptimal, especially for clients that may not be interested in all the received content. This characteristic also incurs a penalty on servers that may need to load and store huge amounts of data from disk to memory.

As we can see, this limitation leads to performance issues, essentially reducing the number of use cases of CRDTs to either devices with virtually infinite memory or the use of small-sized objects. Nevertheless, none of the cases is ideal.

Partial replication is a good approach to mitigate this problem. In partially replicated scenarios, each replica holds a subset of the information of the entire system. Applying this concept to CRDTs would open a window to a more significant range of possible applications and systems that are not possible or supported with classical CRDT replication mechanisms. As an example, consider an application that needs to store a map in the clients' portable device (e.g., a smartwatch). In this scenario, keeping the entire map inside the device is not feasible due to the inherent memory limitations imposed by its small size. Furthermore, the user is usually only interested in a certain part of the map at a given time. This scenario can be modeled using CRDTs, and it is clear how it would benefit from partial replication.

In this section, we survey the work done in the literature related to partial replications of CRDTs. We start by analyzing some existing approaches to the problem with their specific intricacies and limitations and finish up by taking a look at some causally consistent partially replicated systems.

5.1 Partial Replication of CRDTs

To date, not much work has been done that tries to address the problem of partial replication of CRDTs. Current approaches to the problem try to take advantage of the way the CRDT is constructed or even in the semantics of said object. First, in Section 5.1.1, we explore a principled approach to partial replication of CRDTs tailored for objects that hold lots of information and can be separated in smaller parts. In Section 5.1.2, we analyze the issues that arise when composing CRDTs and how to partially replicate them in these scenarios. Lastly, In Section 5.1.3, we take a look at a CRDT synchronization model called non-uniform replication that takes advantage of specific data types query semantics to achieve partial replication.

5.1.1 Conflict-free Partially Replicated Data Types

CRDT objects may become large due to being composed of many elements; the problem only gets worse as the size of each individual element increases. For example, the synchronization of a CRDT set with millions of entries can easily become cumbersome for the system.

Separating the object into smaller sub-objects and replicate them throughout a different set of replicas is a possible solution to this problem. Riak engineers followed a similar approach [35]; however, the proposed solution does not support distributed partial replication and is instead focused on optimizing the local storage at each replica.

I. Briquemont et. al. [12] propose a principled solution to partial replication of CRDTs based on the aforementioned approach. In this work, the authors formulate a novel type of CRDT called *Conflict-free partially replicated data type* (CPRDT) that supports partitioning. Supporting partitioning in CRDTs comes packaged with a set of challenges. First, since some operations might require or affect different parts of data that possibly are not available, one cannot have said operations enabled without adding some pre-conditions [2] to ensure safety and correctness. Second, introducing pre-conditions might interfere with the convergence of the data object; therefore, special care must be taken in this case. Finally, since the parts that are replicated can vary, additional mechanisms need to be added to ensure convergence still happens and no data is lost.

In the CPRDT approach, objects are composed of *particles*. Each CPRDT object can be decomposed into sub-sets called *shard sets*. When originating a partial replica from a CPRDT object, one can specify which parts it is interested in - e.g., in a set CPRDT composed of integers, a partial replica can say it is only interested in the odd numbers stored in the set.

In this system model, given any pair of two replicas, convergence is guaranteed between their common parts, i.e., if an operation affects a *particle* that is not available at a given replica's *shard set*, then it cannot be executed. This type of pre-conditions can be achieved using the functions *required* and *affected*:

[2] See Section 4.2.2

Algorithm 3 Operation-based OR-set with partial replication. Specification taken from [12]

1: **particle definition** *A possible element of the set*
2: **payload** *set S*
3: **initial** $\varnothing$
4: **query** $lookup(element\ e) : boolean\ b$
5: **required particles** $\{e\}$
6: **let** $b = \exists u : (e, u) \in S$
7: **update** $add(element\ e)$
8: **prepare**$(e) : \alpha$
9: **let** $\alpha = unique()$
10: **effect** (e, α)
11: **affected particles** $\{e\}$
12: $S := S \cup \{e, \alpha\}$
13: **update** $remove(element\ e)$
14: **prepare**$(e) : R$
15: **required particles** $\{e\}$
16: **pre** $lookup(e)$
17: **let** $R = \{(e, u) | \exists u : (e, u) \in S\}$
18: **effect**(R)
19: **affected particles** $\{e\}$
20: **pre** $\forall (e, u) \in R : add(e, u)$ *has been delivered*
21: $S := S \backslash R$
22: **fraction**$(particlesZ) : payload\ D$
23: **let** $D.S = \{(e, u) \in S | e \in Z\}$

- ***required(op)*** - This function receives a certain operation *op* and returns the set of particles that are needed by *op* to be properly executed.
- ***affected(op)*** - This function also receives an operation *op* but instead returns the particle that may have its state affected after executing operation *op*.

Once a shard-set has been created, the holding replica cannot become interested in new particles. Additionally, operations are assumed to affect only one particle at a time. To dynamically increase the size of a partial replica, additional protocols need to ensure that any missing data is fetched before the shard-set is ready for usage. While in some cases it is possible to start considering different parts of the object [11], in the general case, it is not feasible without compromising consistency. A simple example is a scenario where operations over the same object affect more than one *particle* and share causal dependencies between them. In this case, starting accepting updates to new particles can result in a causal gap and consequentially, an incoherent state.

To support generality, the system model introduces some assumptions that become more of a hindrance than a help. On the one hand, the shard-set of a replica is assumed not to change over time. On the other hand, all updates are still delivered to all replicas. A more relaxed approach can be achieved through

the use of a centralized model where a root entity maintains all the state of the CRDT object and distributes operations to partial replicas which hold a sub-set of the global state.

As an example to illustrate the previously described model, let us run through the specification for an operation-based OR-set with partial replication (Algorithm 3). As one may recall from Section 4.3.3.4, the OR-set supports both *add* and *rmv* operations and when faced with concurrent *add* and *rmv* operations, precedence is given to the *add* operation. In this specification, apart from the already familiar functions, we can notice the *affected* and *required* functions are being used as preconditions (lines 5, 11, 15, and 19). These preconditions ensure that, in this case, the elements are present in the set before executing the respective operations. Another function that is necessary for the CPRDT model is the $fraction(Z)$ function. This function is used to specify how a new partial replica is created; it receives a set of particles that corresponds to the interest set of the new replica. In this CRDT, since the state is a set, the shard set of the partial replica is the particles that are both in the main replica and the given interest set.

5.1.2 Composing Partial CRDT Replicas

Another reason why CRDTs may become large is that they embed other large CRDT objects inside them. Solving this problem may be done by storing each embedded object separately and hold a reference to them instead. Several issues may appear when considering this type of partial replication model. One may argue that it is possible to represent this type of model in the CPRDT framework; however, for the sake of simplicity, we analyze the concrete case of the riak_dt map to understand what issues may arise.

Riak [36] is a distributed NoSQL key-value store and time-series database that is also highly available. This high availability and partition tolerance comes naturally [6], at the cost of reduced consistency guarantees. Riak provides a library of CRDTs [33], such as Counters, Flags, Sets, among others, that can be used within the data store. One of the included CRDTs is the Riak DT map, a data type that supports composability through the embedding of other composable CRDTs, which includes dt maps themselves.

Even though composition by embedding works well when the composed objects are small in size, performance issues arise as objects get bigger. Meiklejohn [33] addresses this problem of embedding by proposing an alternative composition mechanism through referencing.

While the concept is simple, some issues might occur when we try to update the object or even read from it. One crucial aspect to look after in this type of replication is the way updates are executed to objects that reference other objects. As an example, in Riak's dt map, when referencing a new object, the system needs to guarantee that said object was created in all replicas before assigning a new key to it. Essentially, the underlying system may need to enforce some ordering or atomicity on the execution of the operations [30].

Another issue is related to reads made to references of objects that are not stored in the queried replica. In this case, the system is responsible for recursively retrieving the referenced objects. As described in [33], sometimes it is possible for the referenced object to not be immediately available for fetching. To sidestep this scenario, a pragmatic approach is proposed; the idea is to use the information stored about the type of the object to obtain the bottom value of the said object (i.e., the "default" value with which the specific CRDT is created).

5.1.3 Non-Uniform Replication

The synchronization models we have seen thus far imply that eventually, all updates will reach every CRDT replica. This type of full replication is not necessary for scenarios where the set of queries available for the object does not need the entire state to produce a valid response. Taking advantage of the semantics[3] of the available queries in the interface of an object, one can replicate only the necessary information between replicas and still ensure that the system behaves correctly. This type of replication model is named non-uniform replication and is described in detail in [11].

As an example, the top-k CRDT stores elements which can be ordered and exposes a read operation that returns the top-k elements in the replica. A more concrete example is the top-10 CRDT that could be useful to model a leaderboard for a game. Players write their points to a specific replica, and the top 10 players with the highest point count are returned when the object is queried. In this scenario, replicating all updates might not be necessary, and the system can behave correctly as long as every update that lies in the top 10 (or more generically top k) gets replicated between all replicas. Other designs exist, such as the Top-k CRDT with removes and the Top-K sum CRDT.

In this replication model, two states are said to be equivalent if the results obtained in the execution of any sequence of operations in both states is equal. Additionally, two states are said *observable equivalent* if the result of executing every read-only operation in both states yields the same result. Considering a replicated system in a quiescent state, if for any pair of two replicas, we have that their states are *observable equivalent*, then the system can be considered non-uniform.

At the core of non-uniform replication lies the notion of non-uniform eventual consistency (NuEC). A replication system is said to provide non-uniform eventual consistency if:

(i) Every replica executes all the operations relevant for the final state, called *core* operations. The remaining operations are named *masked* operations.
(ii) All operations must commute.

To close this section, we take a look at an example of a NuCRDT named top-k without removals. The idea behind a top-K object has been presented throughout this chapter, and the top-K without removals has the same characteristics;

[3] The semantics, in this case, refer to the exact meaning of the query, not to be confused with the concurrency semantics mentioned in Section 4.3

Algorithm 4 Specification of the Top-k non-uniform CRDT without removals. Specification adapted from [11]

1: **payload** $elems$: set of $< id, score >$: initial $\{\}$
2: **query get**() : $set\ e$
3:     **let** $e = elems$
4: **update** $add(id, score)$
5:     **prepare**()
6:     **effect**()
7:         $elems = topK(elems \cup \{< id, score >\})$
8: **HasImpact**(op, S) : $boolean\ b$
9:     $R = S \bullet op$
10:     **let** $b = (S \neq R)$
11: **MaskedForever**$(log_{local}, S, log_{recv})$: $set\ of\ operations\ out$
12:     $adds = \{add(id_[1, score_1) \in log_{local}$:
13:         $(\exists add(id_2, score2) \in log_{recv} : id_1 = id_2 \wedge score_2 > score_1)\}$
14:     **let** $out = adds$
15: **MayHaveObservableImpact**$(log_{local}, S, log_{recv})$: $set\ of\ operations\ out$
16:     **let** $out = \{\}$ ▷ Operations are always either core or forever masked
17: **HasObservableImpact**$(log_{local}, S, log_{recv})$: $set\ of\ operations\ out$
18:     **let** $out = \{\}$ ▷ Operations are always either core or forever masked
19: **Compact**(ops) : $set\ of\ operations\ out$
20:     **let** $out = ops$

however, once an element is added, it cannot be removed. The specification can be seen in Algorithm 4.

This CRDT holds a set of K tuples where each tuple holds the information $< id, score >$ (line 1). The update operation (line 4) allows clients to insert the element id in the set with the value $score$. If the element id does not make it to the top-K, it is discarded; therefore, in this scenario, all updates that reach the top-k are considered *core* operations (line 8). Notice, the additional functions Maked-Forever, MayHaveObservableImpact and HasObservableImpact. These functions are responsible for correctly handling core and masked operations. The Masked-Forever function computes the set of *add* operations that have become masked by newer updates - e.g., a new element enters the top-k, therefore, an older element may need to leave the set. The MayHaveObservableImpact (line 21) function computes the masked operations in the replica that have the possibility of influencing other replicas; Since all updates are core or forever masked, this function always returns an empty set. Lastly, function hasObservableImpact computes the set of *add* operations that have not been propagated yet but are currently classified as *core*. In this CRDT, this operation also returns the empty set due to the same reasons as the function MayHaveObservableImpact.

Using this type of replication can bring some storage and bandwidth benefits to the system. On the flip side, this model is limited to very specific data types where the entire state is not needed to give coherent responses to queries. Nonetheless, its applicability in partial replication scenarios is clear, and explor-

ing novel exploitations of data types semantics might be key to solve the problem of partial replication of CRDTs.

5.2 Causally Consistent Partially Replicated Systems

As we have previously seen in Section 4, the causal consistency model is fundamental in the design of CRDTs. While not necessarily obligatory for designing specific CRDTs - e.g., in some cases using weaker models is possible at the expense of having to prove the commutativity of more pairs of concurrent updates; causal consistency makes it easier to reason about the ordering of the events in the system.

On one side, the designs of operation-based CRDTs benefit directly from the guarantees given by causal consistency. On the flip side, since causality is encapsulated inside each state-based CRDT and the entire causal history is propagated between all replicas, communication channels can work under weaker types of ordering guarantees. Under full replication, there exists this trade-off between operation-based and state-based CRDTs; however, when we consider partial replication scenarios, the same trade-off between channel guarantees cannot be considered.

Guaranteeing causal consistency under partial replication is a problem that is deeply tied with the partial replication of CRDTs; having a good foundation on how to achieve it is key to reach a solution that encompasses CRDTs in it. In this section, we take a glance at the reason why efficiently guaranteeing causal consistency under partial replication is hard and how some systems were able to achieve it.

5.2.1 Causal Consistency vs Partial Replication

Achieving causal consistency under full replication has been widely studied [25, 37–45]. These systems implement protocols that are usually efficient when compared with protocols that implement strong consistency models. When transposing to partial replication, using the same techniques may not work; those that do, either require extra metadata to be maintained or further coordination between replicas to obtain said metadata. One way or the other, both cases have scalability problems.

In an ideal scenario, each partial replica would only maintain information regarding values that are currently being replicated by it. Attaining *genuine partial replication* [46] without breaking causality is difficult and might even be impossible without further coordination between replicas or defining a replica communication topology (e.g., Saturn [47]). To give a sense of why causally consistent genuine partial replication is hard to accomplish, consider the scenario displayed in Figure 2. In this example, we consider a distributed system with three replicas ($R1, R2$ and $R3$); $R1$ stores information regarding objects x, z, $R2$ objects x, y and $R3$ objects y, z. Each replica maintains causal information (e.g., a vector clock per object) about the objects it is interested in and only propagates updates to replicas that are interested in said object. $R1$ updates

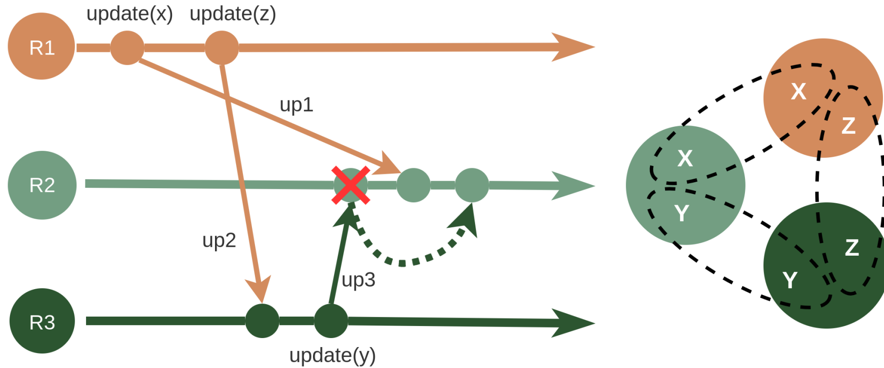

Fig. 2: Events leading to a break in causality in a genuine partial replication scenario

object x (update $up1$) and z (update $up2$) and then propagates the updates to replicas $R2$ and $R3$ respectively. Update $up2$ is causally dependent on update $up1$. If replica $R3$ updated object y (update $up3$) there is a possibility that the information about this update reaches replica $R2$ before update $up1$. Since replica $R2$ does not store any information about object z it would not be able to see the causal dependency that exists between update $up3$ and the in-flight update $up1$ and would mistakenly apply it when it should wait for the arrival of update $up1$.

As we can see, causal consistency under partial replication can raise a different set of challenges that do not happen with full replication.

5.2.2 An Overview of Current Solutions

As seen in Section 3, several mechanisms have been developed to track causality. While efficient in representing the causality dependencies, these mechanisms typically either suffer from a lack of scalability due to the size of the metadata or lose information about causal dependencies, which can lead to false dependencies between concurrent updates. In partially replicated scenarios, keeping the size of the metadata lean is key for the performance of the system. When designing a causally consistent system, these issues need to be taken into account. We now take a look at some approaches that achieve partial replication with causal consistency.

PRACTI [48]: PRACTI is a replicated system that is able to simultaneously achieve partial replication (PR), arbitrary consistency (AC), and topology independence (TI). In this system, replicas can choose each individual object that they will replicate. These objects are then added to the *interest set* of the replica, which is free to change its contents at any given time. Apart from the interest set, each replica maintains a version vector. Updates and reads are executed locally at each replica and propagated afterward. Each update is tagged with the

current value of the node's Lamport clock and the node's ID. PRACTI propagates updates in a similar way to Bayou's [49] log exchange protocol; however, it introduces a novel mechanism; the idea is to separate the propagation of the updates' metadata from the actual values set up by the update. This separation of responsibilities gives PRACTI some freedom regarding the way metadata and data flow through the system. While update messages (named *Body messages*) can be delivered in any order to the replicas, the metadata propagation channels (named *Invalidation streams*) must respect a causal ordering of delivery. Invalidation streams deliver two types of messages: *precise invalidations* and *imprecise invalidations*. Precise invalidations correspond to the metadata regarding single updates. Imprecise invalidations represent multiple ordered precise invalidations and act as a summary of this group of messages. To guarantee correctness, all replicas must maintain all invalidation messages. By doing so, they can control the arrival order of the Body messages and apply them according to the current consistency policy of the system - e.g if the system is guaranteeing causal consistency then replicas need to wait for the arrival of the dependent Body messages before applying one.

Saturn [47]: Saturn is another system that decouples the metadata from the actual data propagation. It is a metadata service for geo-replicated systems; therefore, it focuses only on metadata management and assumes that data is propagated using an existing bulk-data mechanism that fits the application business requirements. Saturn works with small-size metadata that takes the form of *labels*. Labels work as tags for updates and are generated by each server. They can be ordered through their timestamps that are generated from a physical clock at each server. Inside each datacenter exists a centralized component called the *label sink*. This component is responsible for gathering all labels, ordering, and then propagating them to the inter-dc module of Saturn. Clients attach to a certain data center in order to communicate with the system. If a client desires to change datacenters, it must request a migration to Saturn. While Saturn introduces protocols to ensure clients respect causality, the most interesting mechanism lies in how Saturn propagates labels between datacenters that lets it achieve genuine partial replication: *Labels are distributed according to a fixed dissemination tree between datacenters.* In this tree topology, data centers act as leaves, and the nodes leading up to those leaves are servers named *Serializers*. Serializers Inter-communication and communication with the datacenters are done through FIFO channels. This allied to the tree dissemination is sufficient to guarantee causal consistency. Another advantage with the use of a tree is that serializers do not need to propagate labels through branches that will lead to data centers that do not replicate the item associated with the label. This mechanism is fundamental in the sense that it enables genuine partial replication. The use of a tree can raise some challenges. In the case of a failure in a node of the tree, it needs to be recomputed, which is a time-consuming process.Another problem is the fact that every label must be propagated through the tree, which can become a bottleneck in the system. One system that tries to mitigate these issues is C^3 [50]. Like Saturn, C^3 also separates the metadata from the data

propagation; however, the information is propagated directly between every pair of nodes. Not considering a tree raises the need for a larger amount of metadata to be tracked, and genuine partial replication cannot be achieved.

Swiftcloud [51]: A tradeoff that appears to be predominant in causally consistent systems is the throughput at which writes can be applied versus data visibility latency [52]. Swiftcloud provides fast writes and reads at the expense of clients having to access more stale data. In this system, clients maintain a local cache to which they apply updates and later send them to possibly different datacenters. The effect of each update only becomes visible once the operation is replicated in K datacenters (K ¿= 1); in the case of a fault, this allows clients to migrate to other datacenters without being rejected (a rejection would happen in the case where clients read a more up-to-date version of the data and the new datacenter cannot communicate with the faulty datacenter). While clients partially replicate information, data centers are fully replicated. Each client maintains a set of objects in its interest set to which they can write immediately. Datacenters will constantly be pushing notification updates to the clients regarding those objects. Clients, on the other hand, will constantly be sending messages regarding unacknowledged updates to the datacenters. The communication channel is FIFO, and clients must maintain a session opened with the first node they contact. To guarantee convergence and solve conflicts, Swiftcloud's clients use a library of CRDT objects. When clients need to read an object, they do so in a causally coherent state that is exposed by the datacenter. This state may be stale on some occasions, but by exposing a coherent state, the system avoids the possibility of clients generating causal gaps due to the clients' access patterns.

PaRiS [53]: In PaRiS, a transactional causally consistent system that also supports partial replication, clients also maintain a local cache and read from a coherent snapshot computed through the use of a novel causality tracking mechanism named *Universal stable time* (UST). The snapshots are such that they have been installed in every datacenter; therefore, and unlike Swiftcloud, reads do not block. CRDTs may be employed in PaRiS to handle concurrent updates (the default conflict resolution is last-writer wins). Other approaches also rely on clients reading from a stable snapshot [54, 55].

Legion [56]: Legion is a framework that allows client web applications to easily replicate data from the servers and between them in low latency and secure manner. Communication between clients can be done directly without the need to use a server as a middle-man. Since clients' devices are limited in storage, Legion adopts a client-side partial replication approach. In Legion, objects are stored in containers. Each container has an associated multicast group to which clients can join if interested in the objects in the container. Locally, at each client, objects are modified using a library of CRDT objects, and updates are transmitted between nodes through a FIFO channel, which allied to the multicast primitive guarantees causal delivery of updates.

Karma [57]: Karma partially replicates data at the datacenter level. Groups of geographically close datacenters form a consistent hashing ring and in which data is replicated. Karma ensures that causality is guaranteed inside each ring;

however, between ring updates are propagated asynchronously. To avoid clients breaking causality, the system keeps track of in-flight messages and blocks the client from reading from a different ring until the system deems it to be safe.

Discussion: Comparing the different systems, one can notice some characteristics of partially replicated causally consistent systems. Regarding where the partial replication happens, we can divide the systems into two types: client-level and server-level. Client-level systems replicate data on the client's devices; systems such as Swiftcloud, PaRiS and Legion maintain a cache on each client. Server-level systems partially replicate data between the different nodes of the system; The level of replication can go from simple replica servers (PRACTI) to datacenters (Saturn).

A predominant approach in these systems is to let clients read from data that that is potentially stale. Some systems, such as PaRiS and Saturn, expose a consistent snapshot to clients from which they can read. Another approach is the one proposed by SwiftCloud where a client observes a certain update only if it is stored in K different replicas (k-staleness).

Another characteristic of these systems is the topology of the communication of replicas. PRACTI and Legion set no restriction on this topology; however, by limiting the topology of the system, one can introduce some mechanisms (the tree topology in Saturn and the consistent hash rings in Karma) that are not possible in a more generalized scenario.

The separation of the propagation of metadata from the actual data is another interesting mechanism. This approach gives the system more flexibility on how metadata is handled as opposed to other systems that require metadata to be piggybacked inside each message. Systems that take advantage of this technique are PRACTI, Saturn, and C^3.

6 Architecture

In this section, we take a look at our approach to tackling the problem of partial replication with CRDTs. First, let us recap what our goal is and how current methods fail to provide a suitable solution.

As a starting point for our discussion, consider the framework discussed in Section 5.1.1. CPRDTs give us a generic approach to design CRDTs that can be partially replicated; however, this generalization incurs some limitations:

- (i) First, shard sets cannot dynamically increase in size, and trying to do so requires an entirely new replica to be created.
- (ii) Second, to achieve causal consistency, the system assumes the existence of a causal multicast primitive. Doing so requires additional metadata to be stored at each replica and propagated in each message.
- (iii) Lastly, operations are assumed only to affect a particle at a time. For more complex updates, one could divide it into multiple other operations; this, however, may require additional synchronization mechanisms if the affected particles are not replicated locally, and more messages need to flow through the system.

Composition by reference has not been deeply studied. Other approaches, such as non-uniform replication, introduce an interesting replication mechanism; however, current formulations are limited to a particular and somewhat uncommon type of data structures.

Our goal is to design a framework that is able to combine these techniques and provide a way to design CRDTs that can be efficiently partially replicated. The framework aims at reducing the limitations imposed by CPRDTs, ideally allowing replicas to dynamically change the number of shards, the assignment of objects to the shards, or the number of replicas of each shard. An approach based on non-uniform replication could be used to efficiently distribute the objects between replicas without having to propagate redundant data.

Our endeavors will also target designing algorithms that can be used alongside our framework to enforce the consistency of partially-replicated CRDTs. The causal consistency model seems to be the ideal candidate to be considered in these algorithms, as seen throughout Section 5.2.

6.1 System Model

We now discuss the model of the system, which will work as the initial building block for our generalized framework. Current causally consistent systems usually provide either per-object causality or causality for the objects in the system as a whole. In these types of systems, causality tracking and metadata management take up a good portion of the possible system performance - e.g., propagating entire vector clocks in messages become costly as the amount of participating servers increases. Our system tries to find a middle ground between these two types of systems; the idea is to provide causal consistency inside groups of objects but not between groups. The overall system model is depicted in Figure 3.

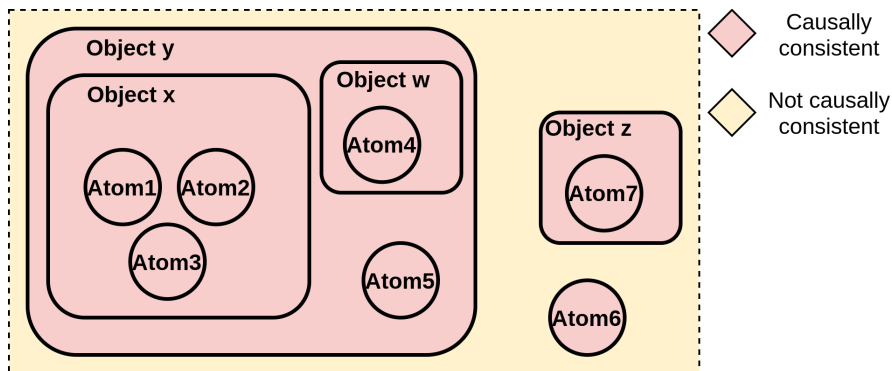

Fig. 3: Example of the organization of atoms, objects and containers

6.1.0.1 ***Atoms and Objects:*** The most basic unit considered in our system are *atoms*. Atoms are opaque and immutable[4] CRDT objects that can be replicated through different replicas in the system. *Objects* are also CRDTs but can be composed of atoms and other objects. Atoms and objects cannot belong to multiple different objects simultaneously; The state held by atoms and objects is guaranteed to be causally consistent at all times. We refer to objects and atoms as *elements*. All elements of the system are globally and uniquely identified. This layout of the system is depicted in Figure 3.

6.1.0.2 ***Operations:*** Operations can be either *queries* or *updates*. Queries can be used to obtain a value from the target element and do not modify the state. Queries may also be used to search for an existing element inside an object. Updates, on the other hand, are used to modify the state of an element. Operations may be applied directly or indirectly to an element. In the case of an indirect operation, it starts at the specified initial object and is then recursively applied through the lower-level objects until it reaches the target element. In the case of atoms, operations can reflect the interface exposed by the underlying CRDT. Regarding objects, operations can also reflect the interface exposed by the underlying CRDT, but they must support two additional operations: *add_element* and *remove_element*. These two operations allow the system to add or remove an arbitrary element to and from an object.

6.1.0.3 ***Nodes:*** The nodes in the system communicate over an asynchronous network. The communication network forms a connected graph that can partition and later recover. Nodes can be either clients or servers. Clients can be thought of as unstable servers that are prone to connect and disconnect more often. Elements may be stored in any node.

6.1.0.4 ***Replication:*** The system distributes atoms in a subset of all replicas ($replicas_{current}$). Atoms can be replicated through a set of potential replicas: $replicas_{potential} = \{r1, r2, ..., rn\}, r_i \in replicas_{all}$ with $replicas_{current} \subseteq replicas_{potential} \subseteq replicas_{all}$. Given an object *obj*, the potential replicas for the object are given by the union of all the potential replicas of the held elements: $obj.replicas_{potential} = \cup el.replicas_{potential} \forall el \in obj.elements$

The way atoms are distributed through the different potential replicas can be specified through policies. Although the role of non-uniform replication is not yet clear, we believe that a great way to take advantage of this mechanism should be in the implementation of these policies.

6.1.0.5 ***Metadata Management*** To implement causality in this system, one could maintain a vector clock for every atom, with the size of the vector being

[4] Immutable in this context means that given a CRDT, one may apply the operations provided by the interface of the abstract CRDT; however, it is not possible to split the atom CRDT into smaller chunks - e.g., splitting the CRDT as in the notion of shard-set presented in the CPRDT framework

the same as the size of the potential replicas set ($replicas_{potential}$) of the highest-level object that contains said atom. With every operation, messages would need to piggyback every vector clock of all elements belonging to the object where the operation was applied. This solution has some serious scalability and performance problems, which might be possible to avoid. Supporting the composition of objects can introduce some levels of hierarchy between them. One can take advantage of this hierarchy to optimize the metadata necessary to keep track of causality. As an example, a particular object could expose a summary of all the lower-level causality tracking devices, which could then be used as a more efficient timestamp. The Coda file system [58] introduced a similar approach with their *volume version vectors*, where these special vectors are used to detect changes in the files of a certain directory.

7 Evaluation

Generalization usually takes a toll on the performance of a system. We intend to evaluate the trade-offs imposed by the mechanisms proposed in our system. We will base our evaluation on several different metrics:

(i) Visibility - This metric states the time it takes for a given operation to become visible for other clients. It is an important metric as the visibility problem is a common trade-off in causally consistent systems.
(ii) Bandwidth usage - When replicas span different datacenters, bandwidth becomes costly. With this metric, we want to see how our system behaves in terms of the number of messages exchanged between replicas and the average size of a message.
(iii) Metadata size and storage usage - Keeping the size of the metadata used in the system minimal is critical for a system to be able to scale. This necessity aggravates when the storage of the devices is limited in size. Measuring this metric will be done either analytically or experimentally.

Apart from these metrics, we also aim at understanding how flexible our system is. To analyze the feasibility of our approach, we will compare our system against various settings of replication and object topologies. On one side, we can populate our system only with atoms; this is equivalent to a system that guarantees causal-consistency per-object. On the other side, we can test against a variation of our system where causality is kept for the system as a whole. Additionally, we can analyze how our system behaves when compared with the systems described in Section 5.2.2.

8 Scheduling of Future Work

Future work is scheduled as follows:

– January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.

– March 30 - May 3: Perform the complete experimental evaluation of the
– May 4 - May 23, 2015: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15, 2015: Deliver the MSc dissertation results.

9 Conclusions

The potential applications of conflict-free replicated data types (CRDTs) in large-scale systems are immense since they avoid explicit synchronization between replicas to decide on the outcome of updates. This property is really desirable in systems where low latency is important.

CRDTs are assumed to maintain their entire state in each replica; however, with CRDTs that store large quantities of data, full replication may be infeasible; one may be forced to shard the CRDT state and let different replicas replicate different shards, i.e., to support partially replicated CRDTs.

In this work, we surveyed current approaches that directly face partial replication of CRDTs as well as systems that support causally consistent partial replication. We laid the foundations for a framework around partially replicated CRDTs. Finally, we presented our evaluation and the schedule of future work.

References

1. Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems. ACM Computing Surveys (CSUR) **49**(1) (2016) 19
2. Herlihy, M.P., Wing, J.M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(3) (1990) 463–492
3. Papadimitriou, C.H.: Serializability of concurrent database updates. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE (1979)
4. Mahajan, P., Alvisi, L., Dahlin, M., et al.: Consistency, availability, and convergence. University of Texas at Austin Tech Report **11** (2011) 158
5. Attiya, H., Ellen, F., Morrison, A.: Limitations of highly-available eventually-consistent data stores. IEEE Transactions on Parallel and Distributed Systems **28**(1) (2016) 141–155
6. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (2002) 51–59
7. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distributed Computing **9**(1) (Mar 1995) 37–49
8. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM **41**(5) (September 1994) 1020–1048
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. 3 edn. Morgan Kaufmann (March 2008)
10. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free Replicated Data Types. Research Report RR-7687, INRIA (July 2011)
11. Cabrita, G., PreguiPreguiça, N.: Non-uniform replication (2017)

12. Briquemont, I., Bravo, M., Li, Z., Van Roy, P.: Conflict-free partially replicated data types. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), IEEE (2015) 282–289
13. Briquemont, I.: Optimising client-side geo-replication with partially replicated data structures. PhD thesis, ICTEAM Institute, Universit catholique de Louvain
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (1978) 558–565
15. Mattern, F., et al.: Virtual time and global states of distributed systems. Citeseer (1988)
16. Fidge, C.: Logical time in distributed computing systems. Computer **24**(8) (1991) 28–33
17. Parker, D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. IEEE transactions on Software Engineering (3) (1983) 240–247
18. Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: In search of the holy grail. Distributed computing **7**(3) (1994) 149–174
19. Kang, B.B., Wilensky, R., Kubiatowicz, J.: The hash history approach for reconciling mutual inconsistency. In: 23rd International Conference on Distributed Computing Systems, 2003. Proceedings., IEEE (2003) 670–677
20. Almeida, P.S., Baquero, C., Fonte, V.: Version stamps-decentralized version vectors. In: Proceedings 22nd International Conference on Distributed Computing Systems, IEEE (2002) 544–551
21. Almeida, P.S., Baquero, C., Fonte, V.: Interval tree clocks. In: International Conference On Principles Of Distributed Systems, Springer (2008) 259–274
22. Almeida, J.B., Almeida, P.S., Baquero, C.: Bounded version vectors. In: International Symposium on Distributed Computing, Springer (2004) 102–116
23. Preguiça, N., Baquero, C., Almeida, P.S., Fonte, V., Gonçalves, R.: Dotted version vectors: Logical clocks for optimistic replication. arXiv preprint arXiv:1011.5808 (2010)
24. Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M.: Logical physical clocks. In: International Conference on Principles of Distributed Systems, Springer (2014) 17–32
25. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ACM (2011) 401–416
26. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA (January 2011)
27. Almeida, P.S., Shoker, A., Baquero, C.: Delta state replicated data types. Journal of Parallel and Distributed Computing **111** (2018) 162–173
28. Enes, V., Almeida, P.S., Baquero, C., Leitão, J.: Efficient synchronization of state-based crdts. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), IEEE (2019) 148–159
29. van der Linde, A., Leitão, J.a., Preguiça, N.: Δ-crdts: Making Δ-crdts delta-based. In: Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data. PaPoC '16, New York, NY, USA, ACM (2016) 12:1–12:4
30. PreguiPreguiça, N.: Conflict-free replicated data types: An overview (2018)
31. : A Conflict-Free Replicated JSON Datatype. IEEE Transactions on Parallel and Distributed Systems **28**(10) (2017) 2733–2746

32. Preguiça, N., Marquès, J.M., Shapiro, M., LeÅ£ia, M.: A commutative replicated data type for cooperative editing. Proceedings - International Conference on Distributed Computing Systems (2009) 395–403
33. Meiklejohn, C.: On the composability of the riak dt map: expanding from embedded to multi-key structures. European Seventh Framework Programme ICT call 10 (2014) 31
34. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In Défago, X., Petit, F., Villain, V., eds.: Stabilization, Safety, and Security of Distributed Systems, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 386–400
35. Brown, R., Lakhani, Z., Place, P.: Big(ger) sets: Decomposed delta crdt sets in riak. In: Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data. PaPoC '16, New York, NY, USA, ACM (2016) 5:1–5:5
36. Technologies, B.: Enterprise nosql database — scalable database solutions — riak. https://riak.com [Online; accessed 03-December-2019].
37. Gokkoca, E., Altinel, M., Cingil, R., Tatbul, E.N., Koksal, P., Dogac, A.: Design and implementation of a distributed workflow enactment service. In: Proceedings of CoopIS 97: 2nd IFCIS Conference on Cooperative Information Systems, IEEE (1997) 89–98
38. Du, J., Elnikety, S., Roy, A., Zwaenepoel, W.: Orbe: Scalable causal consistency using dependency matrices and physical clocks. In: Proceedings of the 4th annual Symposium on Cloud Computing, ACM (2013) 11
39. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Transactions on Computer Systems (TOCS) **10**(4) (1992) 360–391
40. Almeida, S., Leitão, J., Rodrigues, L.: Chainreaction: a causal+ consistent datastore based on chain replication. In: Proceedings of the 8th ACM European Conference on Computer Systems, ACM (2013) 85–98
41. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS), IEEE (2016) 405–414
42. Didona, D., Spirovska, K., Zwaenepoel, W.: Okapi: Causally consistent geo-replication made faster, cheaper and more available. arXiv preprint arXiv:1702.04263 (2017)
43. Gunawardhana, C., Bravo, M., Rodrigues, L.: Unobtrusive deferred update stabilization for efficient geo-replication. In: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17). (2017) 83–95
44. Du, J., Iorgulescu, C., Roy, A., Zwaenepoel, W.: Gentlerain: Cheap and scalable causal consistency with physical clocks. In: Proceedings of the ACM Symposium on Cloud Computing, ACM (2014) 1–13
45. Spirovska, K., Didona, D., Zwaenepoel, W.: Optimistic causal consistency for geo-replicated key-value stores. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), IEEE (2017) 2626–2629
46. Schiper, N., Sutra, P., Pedone, F.: P-store: Genuine partial replication in wide area networks. In: 2010 29th IEEE Symposium on Reliable Distributed Systems, IEEE (2010) 214–224
47. Bravo, M., Rodrigues, L., Van Roy, P.: Saturn: A distributed metadata service for causal consistency. In: Proceedings of the Twelfth European Conference on Computer Systems, ACM (2017) 111–126

48. Belaramani, N.M., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., Zheng, J.: Practi replication. In: NSDI. Volume 6. (2006) 5–5
49. Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Welch, B.: The bayou architecture: Support for data sharing among mobile users. In: 1994 First Workshop on Mobile Computing Systems and Applications, IEEE (1994) 2–7
50. Fouto, P., Leitão, J., Preguiça, N.: Practical and fast causal consistent partial geo-replication. In: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), IEEE (2018) 1–10
51. Zawirski, M., Preguiça, N., Duarte, S., Bieniusa, A., Balegas, V., Shapiro, M.: Write fast, read in the past: Causal consistency for client-side applications. In: Proceedings of the 16th Annual Middleware Conference, ACM (2015) 75–87
52. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: The potential dangers of causal consistency and an explicit solution. In: Proceedings of the Third ACM Symposium on Cloud Computing, ACM (2012) 22
53. Spirovska, K., Didona, D., Zwaenepoel, W.: Paris: Causally consistent transactions with non-blocking reads and partial replication. arXiv preprint arXiv:1902.09327 (2019)
54. Xiang, Z., Vaidya, N.H.: Global stabilization for causally consistent partial replication. arXiv preprint arXiv:1803.05575 (2018)
55. Crain, T., Shapiro, M.: Designing a causally consistent protocol for geo-distributed partial replication. In: Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data, ACM (2015) 6
56. van der Linde, A., Fouto, P., Leitão, J., Preguiça, N., Castiñeira, S., Bieniusa, A.: Legion: Enriching internet services with peer-to-peer interactions. In: Proceedings of the 26th International Conference on World Wide Web, International World Wide Web Conferences Steering Committee (2017) 283–292
57. Mahmood, T., Narayanan, S.P., Rao, S., Vijaykumar, T., Thottethodi, M.: Karma: Cost-effective geo-replicated cloud storage with dynamic enforcement of causal consistency. IEEE Transactions on Cloud Computing (2018)
58. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on computers **39**(4) (1990) 447–459