Large-Scale Geo-Replicated Conflict-free Replicated Data Types

Carlos Bartolomeu
carlos.bartolomeu@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Conflict-free Replicated Data Types (CRDTs) are data types whose operations do not conflict with each other and, therefore, can be replicated with minimal coordination among replicas. Relevant examples of data types that can be implemented as CRDTs are counters and sets. While it is easy to ensure that all replicas of CRDTs become eventually consistent when the system becomes quiescent, different techniques can be used to propagate the updates as efficiently as possible, with different trade-offs among the amount of network traffic generated and the staleness of local information. In this report we study the different alternatives that have been used to propagate updates and discuss potential strategies to automate this selection to optimize the system operation.

1 Introduction

With the advent of cloud computing, and the need to maintain data replicated in geographically remote data centers, searching for strategies to provide data consistency with minimal synchronization became very relevant. Unfortunately, most data types require operations to be totally ordered to ensure replica consistency. This means that operations are diverted to a single primary replica, incurring on long delays and availability problems, or that an expensive consensus protocol such as Paxos[1] is used to order the updates.

Conflict-free Replicated Data Types (CRDTs)[2–4] are data types whose concurrent operations do not conflict with each other and, therefore, can be replicated with minimal coordination among replicas. CRDTs are implemented in such a way that any two concurrent operations A and B are commutative and, therefore, even if they are executed in different sequential orders at different replicas, the final result is still the same. As a result, there are no conflicts among concurrent operations and replicas can often execute operations promptly, without synchronization with other replicas, i.e., operations may be executed locally first and shipped to other replicas only when it becomes appropriated. Using this approach, even if replicas diverge from each other, convergence is eventually reached due to CRDTs properties. Thus CRDTs, unlike other eventual consistency approaches, may strongly simplify the development of distributed application such as social networks, collaborative documents, or online stores[5–7].

Two main types of CRDTs have been proposed, that differ on the techniques they use to reach eventual consistency, namely *operation-based*[2, 8] and *state-based*[2] CRDTs. Operation-based CRDTs send to other replicas the operations that are executed locally; these are later executed remotely also. On the contrary, state-based CRDTs send the full state of the object (which includes the outcome of the operations), such that it can be merged with the local state at remote replicas. Both approaches have advantages and disadvantages as we will see later. A third type of CRDTs has also been proposed more recently, named *delta-based* CRDTs[9], which combines features of the two basic approaches above. Delta-based CRDTs do not ship the full state but, instead, send a smaller state, labeled delta-state, that represents the operations performed between two instants.

The CRDT specification allows for the implementation to choose when it is more appropriate to exchange information among replicas, and allows to postpone eventual consistency to be reached in order to save communication and computation resources. For how long eventual consistency can be postponed depends on the application requirements. These requirements can be captured by a Service Level Agreement (SLA)[10]. By using SLAs, the client or the System Administrator can specify how the system should behave at a given situation.

With this work, we aim at studying the possibility of automating the choice of the CRDT implementation based on the SLA that have been defined and based on the characteristics of the execution environment. These characteristics can be available processing power, observed network latency, timing of propagating messages, etc.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3, we present all the background related with our work. Section 4 describes the proposed architecture to be implemented and Section 5 describes how we plan to evaluate our results. Finally, Section 6 presents the schedule of future work and Section 7 concludes the report.

2 Goals

This work assumes a datastore composed by multiple data centers, where each of them maintains a CRDT replica, and that can be contacted by multiple clients to execute operations. In this context, this work addresses the problem of choosing the best strategy to send the operations/states of each replica to the remaining replicas. In particular, we aim at automatically deciding if it is worth to send a batch of operations (such as in pure operation-based CRDTs) or just the full state of the replica (such as in state-based CRDTs), taking into account the network traffic and the workload of servers.

> *Goals:* After evaluating and comparing the different types of CRDTs, this work focuses on designing a novel, automatic, dynamic strategy that combines the different types of CRDTs in order to optimize CPU costs and/or the communication costs.

Either operation-based, state-based, or delta-based CRDTs have advantages and disadvantages, which will be explained later in the text. An extensive experimental evaluation will determine in which scenario each CRDT implementation performs best. Based on these results, we will design a system capable of dynamically changing the approach, if necessary, to optimize the CPU and network resources while still preserving the consistency requirements of the application, expressed as an SLA.

The project will produce the following expected results.

> *Expected results:* The work will produce i) an implementation of the different CRDT specifications; ii) a system, named A-CRDTs, that automatically combines multiple CRDTs implementation for better performance, iii) an extensive experimental evaluation of A-CRDTs using simulations of workloads.

3 Related Work

This section surveys the relevant work that has been produced in the area of CRDTs. We start by discussing the problem of consistency when managing replicated data, the consistency/cost tradeoff, and how the notion of Service Level Agreement can be used to optimize the system performance. Then we move to introduce the general properties of CRDTs and to present the three main approaches that have been explored to implement that abstraction, namely: operation-based, state-based, and delta-based CRDTs. Finally, we survey some of the most common CRDTs, in particular counter and set datatypes, and compare how their different implementation approaches perform in terms of memory and cpu usage.

3.1 Replication and Data Consistency

When data replicas are placed in geographically distant locations, such that the communication latency among replicas becomes significant, a tradeoff among performance and consistency emerges. In particular, the performance of reads with different consistency guarantees may be substantial. Strongly consistent reads generally involve multiple replicas or must be served by a primary replica, whereas eventually consistent reads can be answered by the closest replica.

Six levels of consistency for read operation have been defined in the literature[10, 11]: strong, causal, bounded, read-my-writes, monotonic, and eventual consistency, as explained below.

- **Strong:** A *read* returns the value of the last preceding *write* performed by any client.
- **Causal:** A *read* returns the value of a latest *write* that causally precedes it or returns some later version. The causal precedence relation $<$ is defined such that $op1 < op2$ if either (a) $op1$ occurs before $op2$ in the same session, (b) $op1$ is a *write* and $op2$ is a *read* that returns the version *writen* in $op1$, or ,by the property of transitivity, (c) for some $op3$, $op1 < op3$ and $op3 < op2$.

- **Bounded(t):** A *read* returns a value that is stale by at most t seconds. Specifically, it returns the value of the latest *write* that completed at most t seconds ago or some more recent version.
- **Read-my-Writes:** A *read* returns the value written by the last preceding *write* in the same session or returns a later version; if no *writes* have been performed to this key in this session, then the *read* may return any previous value as in eventual consistency.
- **Monotonic Reads:** A *read* returns the same or a later version as a previous *read* in this session; if the session has no previous *reads* for this key, then the *read* may return the value of any *write*.
- **Eventual:** A *read* returns the value written by any *write*, i.e. any version of the object with the given key; clients can expect that the latest version eventually would be returned if no further *writes* were performed, but there are no guarantees concerning how long this might take.

In a similar manner, in [11] consistency levels have also been defined for write operations, namely *Writes Follow Reads* and *Monotonic Writes* as described below:

- **Writes Follow Reads:** *Writes* made during the session are ordered after any *Writes* whose effects were seen by previous *Reads* in the session.
- **Monotonic Writes:** *Writes* must follow previous *Writes* within the session.

Although the use of replication raises the problem of data consistency, it has many advantages. In first place, replication offers fault-tolerance. Also, by placing replicas close to the users, replication can provide fast access to data.

The tradeoffs among fault-tolerance and consistency have been captured by the well-known CAP Theorem [12], that states that in a shared-data system we can only have two of the following three properties: Consistency, Availability and Partition-tolerance.

These tradeoffs open a large solution space that has been explored in many different ways by different systems. In our work we weaken consistency, by allowing replicas having different states, to ensure availability and leverage from the CRDTs properties to simplify the replication management.

3.2 Service Level Agreements (SLA)

As we have seen in the previous section, it is possible to define different levels of consistency for read and write operations. Typically, the consistency level that must be enforced is a function of the application semantics and business goals. These requirements can be expressed in the form of a Service Level Agreement (SLA).

In most cases, applications prefer to use the stronger consistency, when the network conditions are favorable. However, when the network is unstable (higher latencies due to congestion, partitions, etc), different applications require different consistency guarantees. Actually, the preferred consistency guarantee may

be even a function of the actual value of the network latency that is observed (for instance, an application may be willing to wait t seconds to get strong consistency but not more).

To cope with the fact that a given application may prefer different consistency levels for different operational conditions, the use of a multiple-choice SLA has been proposed in the context of the Pileus system[10]. This system allows developers to define which level of consistency should be used according to the response time of the system. This means, for a given SLA, if the system predicts that the response time of a strong read is more than the desired, then it should use a other level of consistency, previously specified, in order to achieve the desired response time with more gain. With such an SLA, a system is able to adapt to different configurations of replicas and users and to changing conditions, including variations in network or server load.

3.3 Conflict-free Replicated Data Types (CRDT)

A Conflict-free Replicated Data type (CRDT) is data structure, like a counter or a set, which can be deployed in a distributed system, by placing replicas of the data in multiple servers, in such a way that concurrent operations do not conflict. As a result, a client may contact any server, and execute a sequence of operations on that server, without being forced to wait for (potentially blocking) coordination with other servers.

Since CRDTs avoid explicit synchronisation at every operation, the propagation of updates is often made asynchronously, in background. This means that, at a given time, different replicas may contain different states. However, if the system becomes quiescent, eventually all replicas converge to the same state. However, unlike in other forms of eventual consistency, operations in CRDTs never have to be cancelled or compensated, as a result of the synchronisation among replicas (this derives form the fact that operations never conflict).

The state maintained by each replica is also named the *replica payload*. Client requests are modelled as *operations*. An operation contains the method to be executed and its arguments. The state of a replica after executing a client request is a deterministic function of its state before executing the request and of the operation received. The main difference among CRDT implementations is related with the way different replicas are synchronised. In this respect, there are three main approaches that have been proposed to implement CRDTs: operation-based CRDTs, state-based CRDTs, and an hybrid approach named delta-based CRDTs. These approaches are described in the following sub sections.

Operation-based CRDT: Commutative Replicated Data Type (Cm-RDT) Operation-based CRDTs ensure that replicas eventually converge by propagating all update operations to all replicas. Naturally, operations that are *read-only*, i.e. do not change the state of the object, can be executed locally at any replica, and have the result returned back to the client without the need for any other coordination. Operation-based CRDTs can be seen as implementing a form of active replication, given that all replicas must execute all the requests.

The algorithm used to disseminate the operations among all replicas is independent of the implementation of the CRDT, and several strategies may be used: broadcast, gossip, spanning-trees, etc. However, there are a number of properties that the dissemination process must preserve. In first place, the dissemination must be reliable, such that all operations are received by all replicas and must ensure exactly-once delivery. If a replica does not receive a given operation, or if it applies a given operation more than once, its state may never converge with the state of the remaining replicas. Furthermore, all operation-based CRDTs require causal delivery, as discussed in Section 3.1.

State-based CRDT: Convergent Replicated Data Type (CvRDT) State-based CRDTs ensure that replicas eventually converge by propagating the state of each replica to other replicas and by relying on a *merge* operation that combines the state received from a remote replica with the state of the local replica. When using state-based CRDTs, a request is sent to a single replica, that executes the operation locally. The operation is not propagated to the other replicas. Instead, the state of the replica that has executed the request will, eventually, be propagated and merged with the state of the other replicas.

The key to this approach is to encode the state of the replicas in such a way that the *merge* function becomes idempotent. Thus, if the same state update is applied twice to the same replica, the result should be the same as if it is applied only once. Furthermore, if two different states contain the effects of different, but overlapping, set of operations, the result of the merge function should still be equivalent to the state that would have been achieved by executing each request only once in that replica. To ensure that states can be merged in an idempotent manner is not trivial and, usually, the state must be encoded in a manner that is less space efficient than in operation-based CRDTs.

As before, the approach is independent of the strategy used to decide when to propagate the state of one replica to the other replicas. A new state can be sent every time a client request is processed, or a new state can be sent periodically, and contain the result of many update requests. However, the idempotent property of the merge function puts much less constraints on the dissemination of state updates, when compared to the dissemination of operations. State messages may be delivered more than once or even lost; lost updates are masked by the next state transfer. In the previous case, the order by which state updates is applied is not relevant because the internal structure of the data type together with the *merge* operation ensures convergence of the replicas.

Delta-based CRDT: Conflict-free Replicated Data Type (δ-CRDT) Delta-based CRDTs combine features of operation based and state-based CRDTs. The idea is that, as a result of applying an operation, a *delta-state* is produced. A *delta* only captures changes caused by the associated operation but has the mergeable properties of state-based CRDT. When a *delta A* is merged with other *delta B*, a new *delta* C is created, which represents the *delta A* and *delta B* merged. One delta state is comparable to an operation but has the property of

being able to capture multiple operations as a result of multiple merges. Since a delta states are mergeable, it can be sent to replicas without any requirement because the final state of the replica will always be consistent. All the replicas converge when all the replicas have seen, directly or indirectly, all the deltas states.

3.4 Portfolio of basic CRDTs

In this section we describe and specify some of the basic data types that have been proposed, such as the *counter* and the *set*, which are the basic blocks for more complex data types like graphs. The study of these concrete data types helps in understanding the challenges, benefits, and limitations of CRDTs.

About the algorithms that will be presented, the communication is not described. Instead, we assume the replicas have communications channels and what they send to each other is returned by the operation performed.

Counters A counter is a replicated integer that supports three operations, namely *increment*, *decrement*, and *value* (the first two operations change the state of the counter and the third operation returns its value). It is straightforward to extend the interface to include operations for *adding* and *subtracting* any value. The semantics of the counter are such that its value converges towards the global number of increments minus the number of decrements. A counter is useful in many applications, for instance for counting the number of currently logged-in users.

Algorithm 1 Operation-based Counter

```
1: payload integer i
2:     initial 0
3: function Increment
4:     i := i + 1
5:     return +1                                          ▷ Operation
6: function Decrement
7:     i := i − 1
8:     return −1                                          ▷ Operation
9: function Value
10:    return i
11: function Update(operation k)
12:    i := i + k    ▷ it will increment or decrement depending on k being positive or
    negative
```

Operation-based Counter: An operation-based counter is the simplest CRDT we can find. Its *payload* is an integer i and supports two basic operations: *increment* and *decrement*. It can be extended to support increments of any value as can be

infered just by looking at the specification depicted in Alg. 1. This is possible because any increment or decrement are operations that commute.

Algorithm 2 State-based Grow-only Counter

```
1: payload integer[n] P                                  ▷ One entry per replica
2:     initial [0, 0,...,0]
3: function Increment
4:     r := myID()                                       ▷ r: source replica
5:     P[r] := P[r] + 1
6: function Value
7:     return ∑_i P[i]
8: function Merge(State X)
9:     P[i] := max(P[i], X.P[i]) : ∀i ∈ [0, n − 1]
```

Algorithm 3 Delta-based Grow-only Counter

```
1: payload integer[n] P                                  ▷ One entry per replica
2:     initial [0, 0,...,0]
3: function Increment
4:     r := myID()                                       ▷ r: source replica
5:     P[r] := P[r] + 1
6:     return (r, P[r])                                  ▷ Delta
7: function Value
8:     return ∑_i P[i]
9: function Merge(Delta X)
10:    P[i] := max(P[i], X.P[i]) : ∀i ∈ X
```

State-based Counter: A state-based counter requires a more complex data structure. To simplify the problem, let us specify a grow-only counter. Suppose that we have a payload, like in the operation-based approach, which is an integer i, and the merge operation does the *max* of each payload. Consider two replicas, with the same initial state of 0; at each one, a client originates increment. They converge to 1 instead of the expected 2. Suppose instead that the payload is an integer and that merge adds the two values. This implementation does not have the properties of a CvRDT, given that the merge operation is not idempotent. In [13] the following solution has been proposed to implement a state-based counter: the payload is stored as a vector of integers, with one position per replica (inspired by vector clocks). To increment, each replica adds 1 to its position in the vector. The value is the sum of all entries of the vector. Merge takes the maximum of each entry. The specification can be seen in Alg. 2. To implement a counter that supports *increment* and *decrement* operations we need two vectors,

one for increments and one for decrements. This is because if we use only one vector the max function in the merge operation will not take into account the decrements. The value of the counter is increments minus decrements.

Delta-based Counter: The delta-based counter is inspired by the specification of the state-based counter. After executing an increment it produces a delta which is basically the replica's entry of the vector. In Alg. 3 shows that the payload is the same and the merge operation is the same as well. The only difference in the merge operation is that is most likely for a delta to have the entry of one replica's counter instead of the full state that have the counter's entries of all the replicas. This small difference allow the replicas to send a delta or a small group of deltas merged into one instead of the full state. More details about the performance of this implementation are given in Section 3.6.

Sets A set is a data type that stores elements, without any particular order, and with no repetitions. It has two operations: *add* and *remove*, where *add* adds an element to the set and *remove* removes the element from the set. This data type is the basic block for other complex data structures like maps and graphs as we will see later in this report. So, it is essential to have a specification of this data type for the different approaches of CRDTs. Unfortunately, the semantics of a set under concurrent operations it is not trivial. To introduce the problem, imagine that initially the set contains only one item $\{A\}$ and that its state is maintained by three replicas (1, 2, 3) that are consistent. If, concurrently, replica 1 removes A and then adds A again, replica 2 removes A and replica 3 does nothing, then, there are different serialization orders for these three operations and different orders may provide different final outcomes. A discussion of the possible semantics and valid outcomes of the concurrent set is provided in [2]. In this report we will only consider the *Observed-Removed Set* semantics because it is, at the moment, the best apporach in terms of space and with less limitations.

Operation-based Set: To solve the previous problem using Operation-based CRDTs each element needs a unique tag. Therefore, the payload of the data type will be a set of tuples (*element, tag)*, where the *tag* is a unique identifier associated with each insert operation. This *tag* is needed to support the remove operation: when removing an element, a replica will send to others the element that it wants to remove and all the tags that it sees. This way, when *add* and *remove* operations are concurrent the *add* operation will always win because it is adding an element with a new tag that is not in the set of tags from the *remove* operation. When considering the resulting specification, provided in Alg. 4, the concurrent scenario described before will always has the same outcome, which is a set with the element A. This solution imposes a constraint on the communication pattern: because a *remove* operation always depends on an *add* operation, the operations exchanged between all replicas must respect causal delivery.

State-based Set: The state-base approach requires a more complex solution to support the merge operation. Thus, the payload needs to maintain a causal vec-

Algorithm 4 Operation-based Observed-Removed Set (OR-Set)

payload set S ▷ set of pairs { *(element e, unique-tag u), ...*}
  initial $\emptyset$
function LOOKUP(element e)
  boolean $b = (\exists u : (e, u) \in S)$
  return b
function ADD(element e)
  $\alpha := unique()$ ▷ $unique()$ returns a unique tag value
  $S := S \cup \{(e, \alpha)\}$
  return (add, e, α) ▷ representation of the add operation
function REMOVE(element e)
  $R := \{(e, u) | \exists u : (e, u) \in S\}$
  $S := S \setminus R$
  return $(remove, e, R)$ ▷ representation of the remove operation
function UPDATE(operation (op, e, u)) ▷ executes operations from other replicas
  if $op = add$ **then**
    $S := S \cup \{(e, u)\}$
  if $op = remove$ **then**
    $S := S \setminus u$

Algorithm 5 State-based Observed-Remove Set

payload set E, ▷ E: elements, set of triples { *(element e, timestamp c, replica i)*}
    vect v ▷ v: summary (vector) of received triples
  initial $\emptyset, [0, ..., 0]$
function LOOKUP(element e)
  boolean $b = (\exists c, i : (e, c, i) \in S)$
  return b
function ADD(element e)
  $r := myId()$ ▷ r = source replica
  $c := v[r] + 1$
  $O := \{(e, c', r) \in E | c' < c\}$
  $v[r] := c$
  $E := E \cup \{(e, c, r)\} \setminus O$
function REMOVE(element e)
  $R := \{\forall c, i : (e, c, i) \in E\}$
  $E := E \setminus R$
function MERGE(State B)
  $M := (E \cap B.E)$
  $M' := \{(e, c, i) \in E \setminus B.E | c > B.v[i]\}$
  $M'' := \{(e, c, i) \in B.E \setminus E | c > v[i]\}$
  $U := M \cup M' \cup M''$
  $O := \{(e, c, i) \in U | \exists e, c', i \in U : c < c'\}$ ▷ Old and duplicated elements
  $E := U \setminus O$
  $v := [max(v[0], B.v[0]), ..., max(v[n], B.v[n])]$

tor, where each entry has a timestamp that belongs to a specific replica. The set's payload consists of a set of tuples with three components: *(element, timestamp, replica ID)*. Like the state-based counter, the only information that is sent to other replicas is the state, and the merge operation ensures convergence. The trick in this solution is in the merge operation. An element should be preserved in the merged state only if: either it is in both payloads (set M in Alg. 5), or it is in the local payload and not recently removed from the remote one (set M') or vice-versa (M") - an element has been removed if it is not in the payload but its identifier is reflected in the replica's causal vector. This approach does not impose causal delivery constraints on the communication pattern. The drawback is the space for storage and the cost to send the whole state instead of few operations.

Algorithm 6 Delta-based Optimized Observed-Removed Set

```
1:  payload set E, ▷ E: elements, set of triples { (element e, timestamp c, replica i)}
              set V                          ▷ V: { (timestamp c, replica i)} causal context
2:      initial ∅, ∅
3:  function LOOKUP(element e) : boolean
4:      boolean b = (∃c, i : (e, c, i) ∈ S)
5:      return b
6:  function ADD(element e) : Delta
7:      r := myId()                                              ▷ r = source replica
8:      c := 1 + max({k|(r, k) ∈ V})
9:      E := E ∪ {(e, c, r)}
10:     V := V ∪ {(c, r)}
11:     return ({(e, c, r)}, {(c, r)}                                          ▷ Delta
12: function REMOVE(element e) : Delta
13:     R := {∀c, i : (e, c, i) ∈ E}
14:     R' := V ∩ R
15:     E := E \ R
16:     V := V \ R'
17:     return ({}, R')                                                        ▷ Delta
18: function MERGE(Delta B)
19:     M := (E ∩ B.E)
20:     M' := {(e, c, i) ∈ E|(c, i) ∉ B.V}
21:     M'' := {(e, c, i) ∈ B.E|(c, i) ∉ V}
22:     E := M ∪ M' ∪ M''
23:     V := V ∪ B.V
```

Delta-based Set: The solution for the delta-based set is inspired by the state-base set. This time, instead of the causal vector, we have a set with tuples *(timestamp, replica)* and we call this causal context. The causal context set is used in the merge operation to determine if an element was added or removed from the set. If a replica makes an *add*, then the delta contains the set with the element

added and the causal context has the tag *(timestamp,replica)* associated to that element. If a replica makes a *remove*, then the delta contains an empty set and the causal context has the tag *(timestamp,replica)* associated to element that was removed. In the merge operation their causal contexts are simply unioned; whereas, the new tagged element set only preserves: (1) the triples present in both sets (therefore, not removed in either), and also (2) any triple present in one of the sets and whose tag is not present in the causal context of the other state. This approach is amenable for buffering operations before engaging in communication, because we can compress the result of multiple operations in a single delta state. The drawback is that the causal context will always increase dependently on the number of adds an never decrease. One possible solution to manage the space is by using a distributed garbage collector, that cleans the tags from the causal context of elements that where removed by all replicas. This, obviously, requires some sort of synchronization.

Registers and Graphs There are more examples of CRDTs like Registers and Graphs that have many different applications (for instance, co-operative text editing)[2].

Registers A register is a data type that stores one value in memory. It only has two operations: *assign* - to change the value; and *get* - to obtain the value. The problem to solve, in a distributed register, is the concurrent updates to the register because that operation does not commute. Two major approaches have been identified to address this problem[2]: in the first one operation takes precedence over the other (LWW-Register) and, in the second one, both operations are retained (MV-Register). The *Last-Writer-Wins Register* (LWW-Register) tries to solve concurrency using timestamps to create total order. Timestamps are assumed unique, totally ordered, and consistent with causal order; i.e., if assignment 1 happened-before assignment 2, the former's timestamp is less than the latter's [14]. This way, older updates with lower timestamps are discarded and updates with higher timestamp are preserved. Due to its simplicity, both the operation-based approach or the state-based approach look similar. This means that there are no advantage of one approach over the other in terms of space or cpu consumption. The *Multi-Value Register* (MV-Register), instead of deciding which value is the correct one for concurrent updates, takes all concurrent values and store them in a set. Therefore, it leaves up to the application to decide which value is the correct one.

Graphs A more complex structure is a graph. A graph is a pair of sets (V, E) (called vertices and edges respectively) such that $E \subseteq V \times V$. Any of the Set specifications described above can be used for V and E. Because of its nature ($E \subseteq V \times V$), operations on vertices and edges are not independent. An edge may be added only if the corresponding vertices exist; conversely, a vertex may be removed only if it supports no edge[2]. However, if concurrently a edge is added while a vertex is removed there should exist a deterministic way to preserve

the invariant $E \subseteq V \times V$. There are two intuitive forms to solve this without using synchronization: (i) Give precedence to *removeVertex*(u): all edges to or from u are removed as a side effect. This it is easy to implement, by using tombstones for removed verteces. (ii) Give precedence to *addEdge*(u, v): if either u or v has been removed, it is restored. A good example of a use of a graph is a Replicated Growable Array (RGA). This is a linked list (linear graph) that provides Partial Order to its elements and supports the operation $addRight(v, a)$. Each element (vertex) is connected (edges) to other element. With this concept it is now possible to make co-operative text editing systems. To solve the problem of concurrent adds to the same vertex we can either timestamps like in registers or use other deterministic tie breaker.

3.5 CRDT Use Cases

We now give some examples of how CRDTs can be used in real systems: (i) in a shopping cart solution [15], (ii) in a co-operative text editing [16], (iii) in any system that requires high availability and partitioning[5, 6].

For the shopping cart solution the CRDT set fits perfectly. As it was described before, with CRDT sets we can guarantee that concurrent changes to the set do not conflict with each other, i.e., we can avoid the anomalies of the Amazon shopping cart[15].

For the co-operative text editing we can take the advantages of the Replicated Growable Array (RGA). Because with RGA we can insert elements in a relative position to other elements, we can design a system that supports multiple users writing text in the same page and even in the same word. The system WOOT[16] is one of many systems that do this.

There are other systems that take advantage of CRDTs. For instance, SwiftCloud [5] is able to run transactions in client side due to the mergeable properties of CRDTs.

3.6 Discussion and Comparison

We now discuss the advantages and disadvantages of operation-based, state-based, and delta-based CRDTs, considering the memory consumption, the computational complexity of the operations, and also the metadata garbage collection required by each approach.

Memory Stored in Each Replica Regarding the Counter, we did a small test to compare the memory usage, see Figure 1. This plot depicts the evolution of the memory after N operations in a system with 15 replicas. We compare three implementeations/approaches (i) opearion-based, (ii) state-based, and (iii) delta-based.

We instantly notice that both operation-based and state-based version are constant in memory size. For the operation-based, it does not have any metadata. So in terms of memory it's the same as having a normal centralized counter which

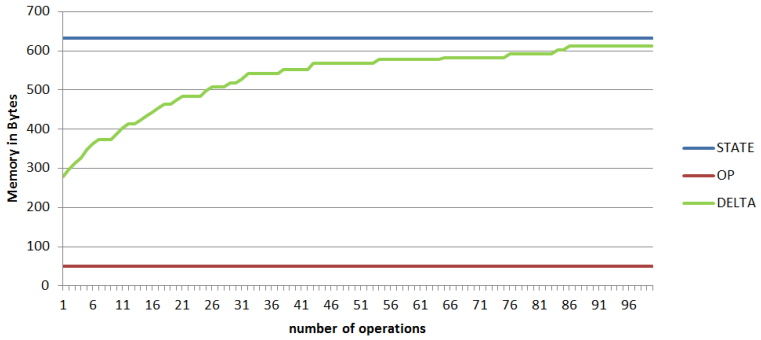

Fig. 1. Comparison of the memory stored in each replica to represent a counter

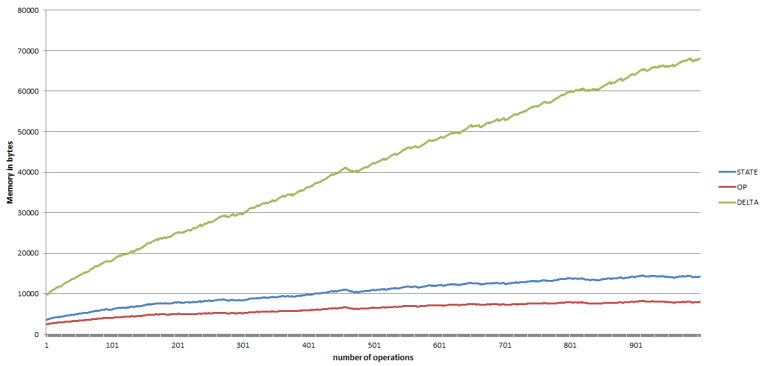

Fig. 2. Comparison of the memory stored in each replica to represent a set

has constant memory size. For the state-based and delta-based approaches, both depend on the number of replicas. However, the state-based depends on the number of replicas that exists in the system, while the delta-based depends on the number of replicas that made an operation. That explains why at the beginning the delta-based uses less memory. But after some time, eventually all replicas will make at least one operation and the memory spent to represent the counter will be the same as if we were using the state-base approach.

Regarding the Set, the memory increases or decreases depending on the number of elements in the set. In particular, the state-base approach has, besides the internal set, a fixed sized vector that depends on the number replicas, and the delta-base approach, instead of a vector, it has a second set that depends on the number of operations (causal context). As we can see in Figure 2, the difference between the memory used in operation-based and state-based is small, because the state-base approach has a fixed sized vector and requires more metadata per element. The delta-based approach has the worst results because the memory also depends on the number of operations, and with no garbage collection it grows quickly.

Complexity of Operations The normal operations of each data type usually do not have a high computational complexity, instead, the merge operation in the state-based and delta-based approach may become expensive. Regarding the counter, the *increment* and *decrement* operations have constant time in any approach. But the merge operation has to make a max function for each entry of the vector and it costs $O(R)$ where R is the number of replicas. Regarding the set, the *add* operation is constant and the for *remove* operation we always have to search for the tags to be removed. When we implement the specifications of the set we can always use some optimizations like hash sets or maps, for fast access to the element and its tags, but we cannot avoid the overhead of the merge operation. The merge operation, either the state-based or delta-based approach, has to make intersection of the payload to see the common elements and then check the elements that are new in both of the payload to not remove them. The state-base approach demands some computational power for this operation comparing to the operation-base approach, and it cost more as the size of the set increases. However, this is not too bad when we compare to the delta-base approach with no garbage collection. Because, in this approach, the causal context grows with the size of the operations made the merge operation takes a huge amount of time searching for a tag in the causal history to know if the element should exists or be removed.

With this analysis we can explain the results obtained (see Figure 3) for the merge operation. The test consists in two replicas with an equal initial state that make different N operations an in the end they converge. For the initial state both replicas made 300 equal operations. Then each replica executed N different operations to simulate divergence. The operations performed were *add* and *remove* with 75% chance of adding, which means the size of the set should increase. In the end we measured the time it took to converge, either by applying all the N operations at once (operation-based approach) or by merging the two states/deltas. The graph shows the time it takes for one replica to apply N operations made by the other replica. As we can see, applying a buffer of operations is not so expensive as making one merge operation, for example, applying 400 operation costs more or less $12\mu s$, one merge of two states that diverge in 400 operations costs $144\mu s$ and one merge of two deltas that diverge as well in 400 operations costs $442\mu s$.

These two results, from memory stored and from complexity, indicate that the operation-based approach is always better. But, if we take into account the memory that takes to buffer the N operations, maybe the other approaches are worth to use.

Memory required for Buffered Operations If in our system we don't want to be constantly communicating with other replicas we should buffer them and then send them all later. In this situation, for the operation-based approach we must keep a buffer of every operation that has not been sent. For the delta-based approach we keep a delta that contains all the operations not sent but not the full state. And for the state-base approach, we cannot buffer any operation so

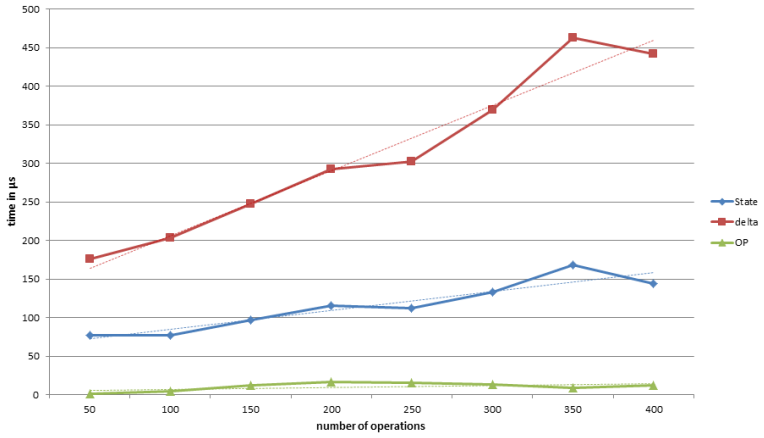

Fig. 3. Time that takes to converge two replicas that diverge in N operations - either by merging states (state and delta) or by applying all operations (op)

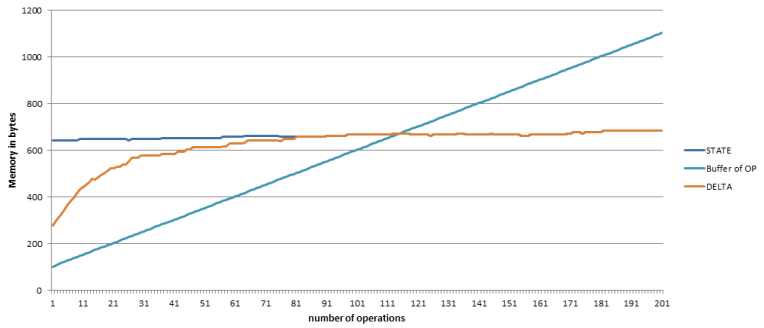

Fig. 4. Comparison between the size of a buffer with N operations and the equivalent stored in a delta and in a state for the counter

we always have to send the full state. As it can be seen in Figures 4 and 5, after buffering some operations the size of the buffer is larger than the size of the actual state (in state-base approach) or the delta, which means the operation-based approach is not always the best option. This test is similar to the first one for the memory stored in each replica. Here we show the size of a buffer/delta/state that is ready to be sent after performing N operations.

Summary To better evaluate the differences among the three approaches, for different data types, we present in Table 1 a summary of the aspects dicussed above.

In terms of memory stored in the replica the operation-based is the best one because it has almost no metadata. On the contrary, the delta-based requires a large amount of memory for storing the causal history, for the set specification.

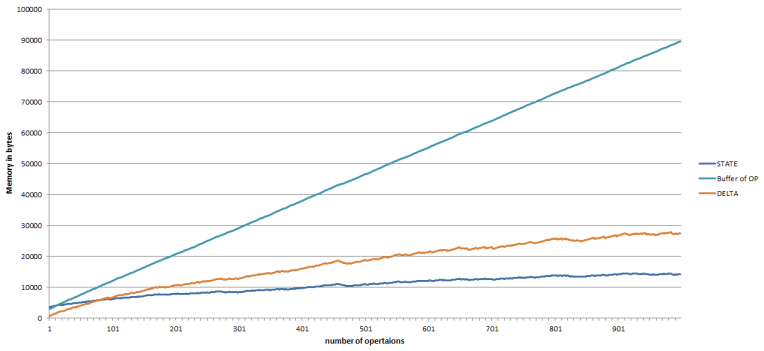

Fig. 5. Comparison between the size of a buffer with N operations and the equivalent stored in a delta and in a state for the set

In terms of computational power, the merge operation has much more overhead when comparing to apply N operations at once due to the complexity of the merge operation. These two factors are where the operation-based approach has big advantages. Now in terms of communication, the operation-based approach must send its operations to every replica and it requires causal delivery. On the other hand, we have the state and delta approaches that do not require causal delivery and only need to send its state/delta to R replicas. The more replicas we send the state, the faster it will converge. In terms of operations that are buffered, once again, the operation-based approach is the worst because it needs an array that grows with the number of operations while the delta-based approach can represent several operations in one object. Finally we have the garbage collector, this is fundamental for the delta-based approach remain useful, otherwise the memory in the replica would grow uncontrollably. We could also use garbage collection in the buffer of operations, in the operation-based approach, to reduce

Table 1. Qualitative comparison between CRDT approaches

Approach	Memory in replicas	computational power	communication	memory of buffers	garbage collection	constrains
operation-based	low	low	contact all replicas	high	low	causal delivery
state-based	average	high	contact R replicas	-	-	none
delta-based	high	high	contact R replicas	average	high	none

its size, i.e., search for operations like $add(A)$ followed by $remove(A)$ that can be removed because they cancel each other.

4 Architecture

Our system, named A-CRDTs, will be a framework that provides to the developers datatypes that are highly available and replicated. Also, they have the chance to define SLAs in order to define how old the data may become and how divergent the replicas can be from each other.

From the point of view of a developer that will use our framework, he will have to decide which datatypes to use, the servers that will be used as replicas and define SLAs for each instance of datatype used. By default, our system will always try to optimize the use of cpu and network, but by using SLAs it will be possible to adapt the system to the desired configuration. The datatypes will be identified with a key, allowing multiple instances of the same datatype. The clients can contact any replica in order to perform *reads* and *writes*, however they should contact the nearest replica for better response times.

About the system itself, Figure 6, each replica will run an instance of the A-CRDTs to store the datatypes and will be able to communicate with other instances in order to converge the state they maintain. In more detail, each server instance will be capable of deciding, in runtime, which approach should be used based on statistics they hold about other replicas and converting datatypes. Each replica has also a monitoring system that gathers information of its resources and from other replicas' monitors. The SLA is unique per instance of a specific data type.

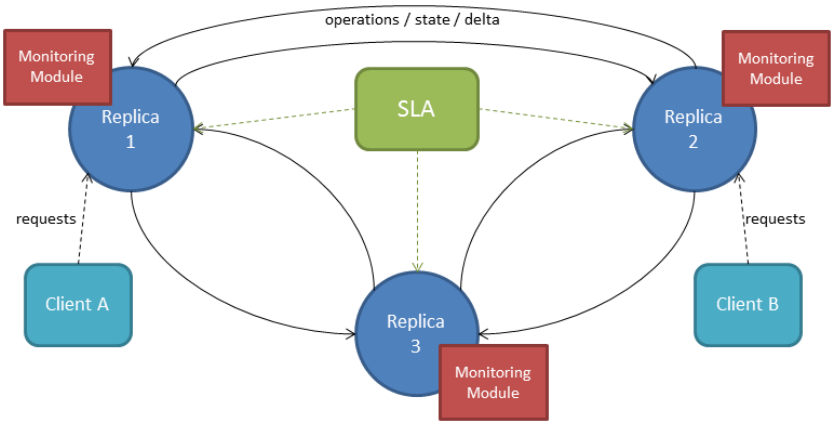

Fig. 6. Architecture of the A-CRDTs framework. - The replicas exchange

4.1 Monitoring

Each replica will have a monitoring module that has access to the machine cpu usage and network usage. The monitoring module is also able to communicate with other monitoring modules in order to gather their information. This information will be:

(i) an estimate of how divergent a replica is from itself, in other words, the number of operations that each replica made differently since the last synchronization;
(ii) the number of operations over time;
(iii) the amount of data sent over time;

Some of this information is already traded while the replicas exchange operations/states, therefore the system will not flood the replicas with messages just to gather this information.

4.2 SLA

Regarding the SLAs, each entry of the SLA has 4 properties: divergence level, network bandwidth, cpu usage and utility. The value of the properties *network bandwidth* and *cpu usage* are the requirements for the system and in return it ensures the correspondent *divergence level*. In this example, Table 2, if at least one of the replicas cannot ensure a *divergence level* of 50 operations with less than 10% *cpu usage* and 10% *network bandwidth* in any CRDT approach, then it should change the *divergence level* to 50 operations. We can achieve this decision by predicting if changing the CRDT approach would match the required resources consumption and assign to it a utility value. Then we multiply that value by the correspondent utility value of the SLA entry. In the end, the SLA entry with the best result contains the *divergence level* that the system should adopt. This concept of using utility values to choose the SLA entry is based on the algorithm described in the D. Terry's paper [10].

Table 2. Example of an SLA

ID	divergence level	network bandwidth	cpu	utility
1	50 op	10%	20%	1
2	50 op	20%	20%	0.8
3	100 op	20%	50%	0.5

4.3 Changing the approach

Changing the CRDT approach in runtime is something that may be expensive, either because it may require some synchronization or because it may cost cpu time to convert the data type itself. However, in a long run, changing the CRDT approach could lead to a better usage of the available system resources. It is not decided yet if all the replicas should change to the same approach or if only one replica changes the approach. Only after some benchmarks will it be possible to decide if it's better to use one strategy over the other or if both are strategies are reliable depending on the workload of the system. Either way, the replica should periodically calculate if the current CRDT approach is the most efficient given the information from the monitoring module and the requirements of an SLA.

5 Evaluation

To test the system, we will create a configurable application that uses the framework to test different workloads of the system. We will also change the running environment of the system by simulating delays in the communication and changing cpu power of the replicas.

5.1 Changes in the environment

Our framework will be useful in the cases that the environment of the global system changes. For example if some communications start to get slower or some replicas start to have too much workload, our system will be able to change the CRDT approach in order to improve the overall performance of the system.

To simulate the network environment we can use a tool called Modelnet [1]. With this tool we can run real applications atop a hypothetical network with assigned bandwidth, latency, queue length, and other link properties. For the cpu usage we can put other applications running at the same time or use a tool like cpulimit [2]. This tool prevents a process from running for more than a specified time ratio, therefore limits the cpu usage of a process.

5.2 Comparison to other systems

Since our system may change the CRDT approach over time, we will perform the same workload tests with a similar system that uses only one CRDT approach. This means that we will test each CRDT approach (state-based, operation-based and delta-based) with different workloads and then compare it to our system.

To properly test the system we will need at least three data centers and a large number clients using the application at the same time. It will all be simulated

[1] http://systems.cs.colorado.edu/2011/04/modelnet-colorado/
[2] https://github.com/opsengine/cpulimit

in a cluster with virtual machines, unless we have the chance to test it in real machines located in different places of the world. The workload tests will be based on the YCSB Yahoo benchmark [17], where we test different proportions of *reads* and *writes*. We will also make another test where we will use different SLAs. On one SLA we vary the 'maximum time without changes' from a low value like 10s to a high value like 100s and on another SLA we vary the 'divergence level' like 50 operations to 500 operations. These are simple examples of SLAs just to analyse how the system reacts to different levels of divergence.

In the end we will measure if the system consumed the least resources for a given SLA. Which means, our system used the most efficient approach given the requirements of the SLA.

6 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 results.
- 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.

7 Conclusions

CRDTs have been used to provide to systems high availability and replication using eventual consistency. Three major approaches have been proposed: operation based, state based and delta based CRDT. Each approach has its benefits. Operation based CRDTs consumes less memory on the server side but requires more communication. On the other hand, state based CRDTs require more space on the server but is good when updates are exchanged with less frequency and require less communication.

So we propose a solution that aims to select the best approach, in runtime, depending on the workload of the system. It also enables the programmers to define SLAs in order to customize the system for specific environments.

We have also presented the architecture of the proposed solution. Its detailed specification, implementation, and experimental evaluation are left for future work, whose schedule has also been presented.

Acknowledgments We are grateful to Manuel Bravo for the fruitful discussions and comments during the preparation of this report. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/2013, and via the project PEPITA (PTDC/EEI-SCR/2776/2012).

References

1. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2) (May 1998) 133–169
2. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506 (January 2011)
3. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. SSS'11, Berlin, Heidelberg, Springer-Verlag (2011) 386–400
4. Letia, M., Preguiça, N.M., Shapiro, M.: Crdts: Consistency without concurrency control. CoRR **abs/0907.0929** (2009)
5. Zawirski, M., Bieniusa, A., Balegas, V., Duarte, S., Baquero, C., Shapiro, M., Preguiça, N.M.: Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. CoRR **abs/1310.3107** (2013)
6. Navalho, D., Duarte, S., Preguiça, N., Shapiro, M.: Incremental stream processing using computational conflict-free replicated data types. In: Proceedings of the 3rd International Workshop on Cloud Data and Platforms. CloudDP '13, New York, NY, USA, ACM (2013) 31–36
7. Preguica, N., Marques, J.M., Shapiro, M., Letia, M.: A commutative replicated data type for cooperative editing. In: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems. ICDCS '09, Washington, DC, USA, IEEE Computer Society (2009) 395–403
8. Baquero, C., Almeida, P., Shoker, A.: Making operation-based crdts operation-based. In Magoutis, K., Pietzuch, P., eds.: Distributed Applications and Interoperable Systems. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2014) 126–140
9. Almeida, P.S., Shoker, A., Baquero, C.: Efficient state-based crdts by delta-mutation. CoRR **abs/1410.2803** (2014)
10. Terry, D., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M., Abu-Libdeh, H.: Consistency-based service level agreements for cloud storage. In: Proceedings ACM Symposium on Operating Systems Principles, ACM (November 2013)
11. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.W.: Session guarantees for weakly consistent replicated data. In: Proceedings of the Third International Conference on Parallel and Distributed Information Systems. PDIS '94, Washington, DC, USA, IEEE Computer Society (1994) 140–149
12. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. PODC '00, New York, NY, USA, ACM (2000) 7–
13. Bieniusa, A., Zawirski, M., Preguiça, N.M., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: An optimized conflict-free replicated set. CoRR **abs/1210.3368** (2012)
14. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565
15. 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. SIGOPS Oper. Syst. Rev. **41**(6) (October 2007) 205–220

16. Oster, G., Urso, P., Molli, P., Imine, A.: Data consistency for p2p collaborative editing. In: Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work. CSCW '06, New York, NY, USA, ACM (2006) 259–268
17. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing. SoCC '10, New York, NY, USA, ACM (2010) 143–154