Conflict-free Partially Replicated Data Types

Iwan Briquemont*, Manuel Bravo*†, Zhongmiao Li*† and Peter Van Roy*
*Université catholique de Louvain, Belgium
†Instituto Superior Técnico, Universidade de Lisboa, Portugal

***Abstract*—Designers of large user-oriented distributed applications, such as social networks and mobile applications, have adopted measures to improve the responsiveness of their applications. Latency is a major concern as people are very sensitive to it. Geo-replication is a commonly used mechanism to bring the data closer to clients. Nevertheless, reaching the closest datacenter can still be considerably slow. Thus, in order to further reduce the access latency, mobile and web applications may be forced to replicate data at the client-side. Unfortunately, fully replicating large data structures may still be a waste of resources, specially for thin-clients.**

We propose a replication mechanism built upon conflict-free replicated data types (CRDT) to seamlessly replicate parts of large data structures. We define partial replication and give an approach to keep the strong eventual consistency properties of CRDTs with partial replicas. We integrate our mechanism into SwiftCloud, a transactional system that brings geo-replication to clients. We evaluate the solution with a content-sharing application. Our results show improvements in bandwidth, memory, and latency over both classical geo-replication and the existing SwiftCloud solution.

I. Introduction

Globally accessible web applications, such as social networks, aim to provide low-latency access to their services. Thus, data locality is a fundamental property of their systems. Geo-replication is a common solution where data is replicated in multiple datacenters [1]–[3]. In this scenario, user requests are forwarded to the closest datacenter. Therefore, the latency is reduced. Unfortunately, the latency, even when accessing the closest datacenter, may still be considerable. [4], [5] argue that clients are sensitive to even small increases of latency.

Systems such as [6], [7] use caching techniques to yet reduce latency even more. However, this can be challenging and expensive. For instance, one could simply use client caches for reading purposes. Nevertheless, in order to keep some consistency guarantees and freshness of data, mechanisms, such as cache invalidation, need to be used. Scaling these kinds of techniques is difficult and directly affects the performance. Moreover, one could let clients apply write operations locally and eventually propagate them. However, this can cause conflicts between replicas and potential rollbacks.

The recently formalized CRDTs [8], [9] can serve to diminish the impact of some of the previously mentioned problems. These data types are conflict-free by default; therefore, no conflict resolution mechanisms need to be written by application developers. SwiftCloud [10], a geo-replicated storage system that ensures causal consistency, benefits from CRDT semantics. It replicates CRDTs not only across datacenters, but it also replicates them in clients. It allows read and write operations to be directly executed in clients caches. In consequence, SwiftCloud reduces latency, and enables off-line mode during disconnection periods.

The current specifications of CRDTs do not allow partitioning. Thus, a CRDT replica is assumed to contain the full data structure. We believe partitioned CRDTs may pose several benefits for current applications. First, CRDTs can easily become heavy data structure, such as a set CRDT that contains the posts of a user wall in a Facebook-like application. In many cases, users are simply interested in the most relevant posts, according to some criterium. For instance, one may be interested in reading the top-ten most voted posts of a Reddit-like application. Thus, replicating the whole CRDT is a waste of resources, of both storage and bandwidth. The former can be critical when thin devices, such as smartphones, are considered as clients. These types of clients have limited memory resources; therefore, it is convenient to avoid storing unnecessary data. On the other hand, bandwidth is one of the most costly resources offered by cloud providers such as Amazon S3 [11], Google Cloud Storage (GCS) [12], and Microsoft Azure [13]; therefore, it is beneficial to use it efficiently. Second, the full replication of CRDTs in clients may arise security concerns. By partitioning CRDTs, applications could precisely decide which data each client stores. This could keep malicious clients from storing sensitive data. Finally, they can also be used to provide multiple fidelity requirements for data accommodated in resource-limited devices, while keeping consistency between fidelity levels [14]. This could be achieved by not replicating less important information on mobile devices.

In this paper, we propose a new set of CRDTs that allows partitioning. We call them "Conflict-free Partially Replicated Data Types" (hereafter CPRDTs). We study how partitions of the same CRDT can interact among each other and still maintain its consistency guarantees. Furthermore, we revise previously defined CRDT specifications and propose new specifications that consider partitioning. One could claim that developers could simply re-format their data structures to obtain similar benefits; nevertheless, this adds complexity to application development and, in some cases, optimal results can be difficult to achieve. We propose to better integrate CPRDTs into the system. Thus, developers will benefit from them transparently, without being aware of their existence.

The major contributions of this paper are the following: (i) definition of the new CPRDTs, which includes revisiting the specifications of previously defined CRDTs; (ii) extension of SwiftCloud to integrate CPRDTs; and (iii) extensive evaluation of the performance improvements of CPRDTs in SwiftCloud. The latter includes the implementation of a Reddit-like [15], [16] application, called SwiftLinks, on top of SwiftCloud.

The remainder of the paper is organized as follows: Section

II presents a formal definition of the partitioned CRDTs; Section III discusses some practical issues of CPRDTs; Section IV presents an extensive evaluation of the SwiftCloud extension that includes CPRDTs; Section V briefly describes preceding related work; finally, Section VI concludes the paper.

II. Conflict-free Partially Replicated Data Types

Allowing partitioning poses new challenges: all operations are not enabled on partial replicas, which means new preconditions must be added to ensure correctness. However, these preconditions must not compromise the convergence of replicas. Plus, a partial replica could vary the parts it keeps, by choosing to replicate more, or less, parts. This has to be done without losing data and still achieving convergence.

A. Example of use

Let us use an example to illustrate the advantages of CPRDTs: the user wall of a social network. We can model a user's wall as a set. In this example, there are four users that interact: Alice, Bob, Charlie and an anonymous user. Bob is a friend of Alice, while Charlie is a friend of Bob, but not of Alice. Participating users may want to read or post something in Alice's wall. We make two assumptions: (i) users maintain a full replica of their wall; and (ii) a user X that reads or posts in user Y's wall replicates user Y's wall locally.

Each post contains a date, an author, and a message. Each user is allowed to read a subset of other users' walls, depending on their friendship and posts visibility (private or public). For instance, Bob can read all the posts of Alice's wall because of their friendship. Nevertheless, Charlie can only read public and Bob's posts (friends of friends). Finally, any other user can only read public posts.

We can assume that Alice has been using the social network for a few years and there are a considerable number of posts on her wall. It seems natural that a user should not have to replicate the whole wall to simply read the latest posts. Nevertheless, this is what presumably may occur in a fully-replicated scenario (CRDT-like), where the data structures cannot be partitioned and we still want to replicate data in clients-side. One solution is to manually split the data structure according to some criteria (e.g. by date, author or privacy setting). However, developers then need to anticipate how users will use the application. While possible in some cases, it seems to make the application cumbersome to write.

In this scenario, CPRDTs have two applications. On the one hand, CPRDTs abstract the partitioning from the application. Thus, from the point of view of programmers, there will only be one logical data structure per wall. We strongly believe this may ease developers task. Moreover, this allows a more efficient and fine-grained partitioning adapted to the needs of a particular client in a specific point of time, which may be impossible if the partitioning is done manually by developers. The second application of CPRDTs is related to the enforcement of security policies. We may want users to only replicate posts that they are allowed to see. This could keep malicious users from storing sensitive data locally.

B. Definitions

Before defining CPRDTs, we have to clarify some concepts that we will use throughout the paper. An *object* is a named instance of a CRDT or CPRDT in our case. Each participating process replicates a set of objects. A process that replicates an object is called *replica* of the CRDT (or CPRDT) instantiated by the object. Objects can be read using *query* operations and modified by *update* operations. Query operations return the abstract state of the object, that we call the *data* of the object. Nevertheless, additional data, which we refer as *metadata*, is kept internally to ensure convergence.

An update operation can have preconditions that capture its safety requirements. In consequence, an operation is said to be *enabled* at a replica, if it satisfies its preconditions. For instance, the remove operation of a set is enabled only if the element to be removed is present in the set.

Previous definitions fit into both CRDTs and CPRDTs. Nevertheless, for CPRDTs, we further consider that a process might replicate an object partially: it only has access to a part of data, thus the process only keeps the metadata required for that given part. Intuitively, this means that only part of the data structure is replicated: some elements of a set, a subgraph of a graph, or a slice of a sequence. CRDTs that only have one element, such as counters and registers, can not be partitioned and therefore do not need to be specified as CPRDTs.

particle We define a *particle* as an element of a collection. For instance, a particle in a set would be any element that can be added to the set.

Apart from the definition of particle, we introduce three new concepts: *shard set*, *required*, and *affected*.

shard set Each replica of a CPRDT x_i has associated a set of particles, namely *shard set* in analogy to the databases concept. Respectively, $\mathrm{shard}(x_i)$ is a function that returns the *shard set* of x_i. A replica x_i only knows the state of the particles in $\mathrm{shard}(x_i)$; therefore, it can only enable query operations that require those particles. Furthermore, a replica x_i only needs to receive update operations that affect the particles in $\mathrm{shard}(x_i)$.

There are two special cases: a *full* replica and a *hollow* replica. We use π to represent the full set of possible particles a CPRDT may be interested in. Thus, we say that a *full* replica's *shard set* is equal to π. Notice that a *full* replica CPRDT is equivalent to a normal CRDT. On the other hand, when $\mathrm{shard}(x_i) = \varnothing$, then x_i is a *hollow* replica (as named in [17]). A *hollow* replica does not maintain any state. Nevertheless, it can still handle updates (section II-C2).

required For an operation *op* with its arguments, $\mathrm{required}(op)$ is the set of particles needed by *op* to be properly executed. This means that, for replica x_i, an operation *op* is enabled only if $\mathrm{required}(op) \subseteq \mathrm{shard}(x_i)$. For instance, for the lookup operation of a set, $\mathrm{required}(lookup(e)) = \{e\}$ where e is an element of the set. In case $e \notin shard(x_i)$, the replica x_i does not know whether e is in the set because it has not kept a state for it. This implies that updates affecting e have not been necessarily seen by x_i.

affected The function $\mathrm{affected}(op)$ returns a particle that may have its state affected after executing an update operation. We assume that an update can only affect one particle. This may

not be true for complex data structures, however it is always possible to split an operation into several ones that each only affects one particle. For example, for a graph, an operation for removing a vertex will remove the vertex as well as all its edges. It can be split into several sub-operations that firstly remove all edges of the vertex and then remove the vertex.

C. Replication

As for the original CRDTs, we consider two equivalent replication techniques: state- and operation-based. Allowing partitioning introduces changes in the way these replication techniques work. Furthermore, concepts such as causal history and convergence have to be revisited. The following definitions are based on the ones in [8] for fully-replicated CRDTs.

To simplify our definitions, we assume that the *shard set* of a CPRDT is fixed. However, in practice, it can be necessary to dynamically change it. Nevertheless, definitions apply if we consider that changing the *shard set* is equivalent to the creation of a new CPRDT replica.

Since the abstract state of a CPRDT may change after applying an update, we denote the abstract states of a CPRDT replica (x_i) by an increasing numbered sequence as $s_k(x_i)$, such as $s_0(x_i), s_1(x_i)... \ s_k(x_i)...$

Now we define when two replicas are equivalent.

Definition 1 (Equivalence)**.** *x_i and x_j have equivalent abstract states if all* `query` *operations q, for which* $\text{required}(q) \subseteq (\text{shard}(x_i) \cap \text{shard}(x_j))$*, return the same values.*

Different replicas of the same CPRDT might have different *shard sets*. Thus, we define intersecting abstract state as the abstract state for the particles in the intersection of *shard sets*.

Definition 2 (Intersecting abstract state)**.** *For a replica x_i with its current state $s_k(x_i)$, $s_k(x_i|x_j)$ denotes the s_k state for particles $\in \text{shard}(x_i) \cap \text{shard}(x_j)$.*

The requirement for replicas to converge is that they apply, directly or indirectly, the same update operations. We can informally define the causal history of a replica, denoted by $C_k(x_i)$, as a set containing the applied update operations. As x_i applies each operation, its causal history goes through a sequence of states $C_0(x_i), C_1(x_i), ..., C_k(x_i), ...$ We also define the intersecting causal history as $C_k(x_i|x_j) = \{f \in C_k(x_i) | \text{ affected}(f) \in (shard(x_i) \cap shard(x_j))\}$. Intuitively, it includes updates from $C_k(x_i)$ that affect the particles of x_j.

Now, we are ready to formally define convergence in the context of CPRDTs:

Definition 3 (Eventual Convergence of Partial Replicas)**.** *Two partial replicas x_i and x_j of an object x converge eventually if the following conditions are met:*

- *Safety: $\forall i, j : \forall k, k'$, if $C_k(x_i|x_j) = C_{k'}(x_j|x_i)$, then $s_k(x_i|x_j) = s_{k'}(x_j|x_i)$.*
- *Liveness: $\forall i, j : \forall k$, if $f \in C_k(x_i)$ and $\text{affected}(f) \in \text{shard}(x_j)$, then $\exists k'$ that $f \in C_{k'}(x_j)$.*

1) State-based partial replication: In this replication technique, a replica ships its whole internal state to the rest. Upon arrival, replicas merge both the local and the received states. The merge method is an idempotent, commutative and associative operation that has two replicas internal states as arguments. In the CPRDTs context, the *merge* method used by a replica must only merge the state of the particles belonging to the intersection between local and remote replicas *shard sets*, and ignore the rest.

State-based replication is an interesting propagation mechanism since it poses almost no communication requirements. Nevertheless, it may be expensive to always ship the full internal state. CPRDTs can optimize this technique since only parts of the state need to be sent and received. We define the causal history of a replica for state-based replication as follows:

Definition 4 (Causal History on Partial Replicas - state-based)**.** *For any replica x_i of x:*

- *Initially, $C_0(x_i) = \varnothing$.*
- *Before executing* `update` *operation f,*
  if $\text{affected}(f) \in \text{shard}(x_i)$ then execute f and $C_{k+1}(x_i) = C_k(x_i) \cup \{f\}$,
  otherwise $C_{k+1}(x_i) = C_k(x_i)$.
- *After executing merge against states x_i, x_j, $C_{k+1}(x_i) = C_k(x_i) \cup \{f \in C_{k'}(x_j) | \text{ affected}(f) \in \text{shard}(x_i)\}$*

To achieve convergence with state-based replication on partial replicas, updates operations cannot be applied if it affects a particle that is not in that replica's *shard set*. This would violate the liveness property of convergence as that update might not be added to the causal history of another replica when merging. Thus, an operation f is disabled if $\text{affected}(f) \notin \text{shard}(x_j)$. On the other hand, since the replicas only converge on their common parts, a replica x_i just needs to send to another, x_j, the state of the intersection of their shards ($\text{shard}(x_i) \cap \text{shard}(x_j)$).

2) Operation-based partial replication: As with classical CRDTs, the update operations are divided into two phases: *prepare* and *downstream* phase. The former is done at the source replica and does not have any side-effect. The latter is applied at all replicas and it affects the state of the replica. We define the causal history of a replica for operation-based replication as follows:

Definition 5 (Causal History on Partial Replicas - op-based)**.** *For any replica x_i of x:*

- *Initially, $C_0(x_i) = \varnothing$.*
- *After executing the* `downstream` *phase of operation f at replica x_i,*
  if $\text{affected}(f) \in \text{shard}(x_i)$ then $C_{k+1}(x_i) = C_k(x_i) \cup \{f\}$,
  otherwise $C_{k+1}(x_i) = C_k(x_i)$.

In contrast to CRDTs, CPRDTs only have to broadcast updates to the replicas interested in the particles affected by the update. Therefore, an update u is broadcasted to x_i if $\text{affected}(u) \in \text{shard}(x_i)$. This poses an interesting situation.

A CPRDT replica can sometimes complete the first phase of the update operation without necessarily completing the second phase. For instance, a replica x_i, whose $\mathrm{shard}(x_i)$ are particles a and b, receives an update operation that affects particle c. In this situation x_i may complete the prepare phase, broadcast the downstream operation to the interested replicas, and discard it locally. We name this scenario *blind updates*. This can only happen in operation-based replication. Hollow replicas, whose shard is empty, can only do blind updates.

D. Specification of CPRDTs

In this section, we present the specifications of an operation-based observed-remove set (OR-set) CPRDT. We resort into this example in order to better illustrate how to integrate the newly defined concepts into a CRDT (original specifications in [8]); and thus, transform it into a CPRDT. More examples of CPRDTs and generic specification templates, for both operation- and state-based, are found in [18].

An OR-set works as follows: (i) elements are uniquely tagged by the source replica when added to the set. The source replica is the one receiving the client operation. (ii) concurrent additions of the same element are all reflected in the set internal state by storing them with different tags. (iii) a remove operation is transformed into the list of unique tags related to the element to be removed that are present in the source replica. Since causal delivery is assumed, this ensures convergence of replicas even in the presence of concurrent adds and removes of the same element.

The specifications incorporate (i) the particle definition (line 1); (ii) the *required* and *affected* preconditions (lines 11, 15 and 19); and (iii) a new function called *fraction*. The *fraction* operation allows us to create new partial replicas from a subset of a given replica. The subset we want to copy in the new replica is defined by a set of particles. More formally, *fraction* can be defined as follows:
$x_j = \mathrm{fraction}(x_i, Z)$, where Z is the set of particles to replicate. The operations ensures that $\mathrm{shard}(x_j) = \mathrm{shard}(x_i) \cap Z$.

Specification 1 Op-based OR-set with Partial Replication

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

III. Practical issues

In this section, we discuss (i) *shard queries*, and (ii) the implications of allowing dynamic shard sets. Both issues are relevant for making CPRDTs practical.

A. Shard queries

The operation *fraction*, introduced in II-D, is the canonical form to define the *shard set* of a replica. Nevertheless, *fraction* is not practical. In practice, applications will transform their semantics into a high-level query language. For instance, an application could issue a query in the form of "give me the first 10 elements of your sorted set". We name this type of queries *shard queries*. They bridge the gap between the application semantics and the function *fraction* adding expressiveness to the usage of CPRDTs.

There are two types of *shard queries*: version-independent and version-dependent. The former only depends on the properties of the particles, and not in the version of the CPRDT. In contrast, the latter depends on the current version of the CPRDT. Let us use a CPRDT set whose domain is the set of integers as example. A version-independent query could be "integers greater than 0". This *shard query* will always return the same *shard set* ($(0, +\infty)$) independently of the queried CPRDT version. On the other hand, a version-dependent query, such as "the 10 highest integers in the set", will return a different *shard set* depending on which elements have been already added, and removed, on the version being queried.

Version-independent queries are easier to work with: they are comparable. One could determine which query is more specific without knowing the version of the CPDRT they apply to. While with version-dependent queries, one can only compare queries if they apply to the same version. Nevertheless, both types are needed in order to make CPRDTs practical.

B. Dynamic shard set

Dynamic *shard set* refers to the capability of a partial replica to modify, either shrink or expand, its *shard set*. We believe this capability is useful in practice, e.g. a client may become interested in new parts. Having dynamic *shard set*, a replica does not need to be re-created, only the missing state needs to be grabbed. Nevertheless, maintaining convergence in some scenarios can become challenging.

On the one hand, a partial replica can easily shrink its *shard set* without compromising convergence in the operation-based scenario. A replica only needs to take two things into consideration: (i) updates prepared locally have been already broadcasted, and (ii) the data to be dropped is replicated by some other replica; therefore, data is not lost. On the other hand, expanding a partial replica is more tricky. For instance, in an operation-based scenario, the following situation can easily occur: (i) a replica's (x_i) *shard set* is a, c; therefore, x_i does not receive updates that affect b; (ii) suddenly, x_i becomes interested in b and starts accepting updates on b; (iii) unfortunately, x_i will not converge since updates have been missed. In order to ensure convergence, extra communication between replicas would be needed to recover dropped updates. This would add complexity to the underlying protocols.

In state-based replication, shrinking or expanding the *shard set* is simpler. On the one hand, a replica only needs to broadcast its state before shrinking its *shard set*. On the other hand, a replica that wants to expand its *shard set* only needs to merge its current state with the state of a replica that contains new particles.

IV. Evaluation

In this section, we report the results of our experimental evaluation. This study aims at evaluating the benefits of CPRDTs in terms of memory, bandwidth and latency.

SwiftLinks In order to evaluate our solution, we implemented an application, namely SwiftLinks, on top of SwiftCloud. SwiftLinks is a vote-based content-sharing application based on Reddit. In short, the application allows users to create forums where they can publish posts. Then, users can vote posts positively or negatively. As a consequence, posts get ranked. In addition, users can comment posts and other comments. Users can also vote comments, and consequently, comments get ranked (more information [15], [16]).

SwiftLinks is modeled with three types of data structures: (i) OR-Set for each forum, (ii) a novel Remove-once Tree for each tree of comments, and (iii) Last-Writer-Wins Registers for each vote associated to a post/comment. The application uses both types of queries: version-independent and version-dependent. The former is mostly used for reading single comments or posts. The latter is used for reading ranking of posts and comments.

Warm-up We used Reddit's API to fetch data to warm up our system. For each benchmark, we create 10000 posts over 20 forums (so an average of 500 posts per forum). Each post has 20 comments on average. Moreover, each post has an average of 170 votes, while comments an average of 13 votes.

Workload Our workloads are composed by read and update operations. Read operations are executed over posts and comments. On the other hand, there are three types of update operations: (i) new post, (ii) new comment, and (iii) new vote.

For most of the experiments, 20% of the operation are updates and 80% are read operations. Furthermore, 90% of the operations are biased to previously accessed objects. This means that they are likely to hit the cache. The rest (10%) is done on randomly selected posts and comments.

A. Integration of CPRDTs into a real system

We chose SwiftCloud [10] to integrate CPRDTs. SwiftCloud is a geo-replicated cloud storage system written in Java that stores CRDTs and caches data at clients. It consists of several datacenters that fully replicate the key-space. Clients indirectly communicate through the datacenters. In absence of failures, a client always interacts with its closest datacenter and caches accessed data in its local cache. SwiftCloud provides transactional causal+ consistency. Transactions are first executed and committed on the client side, then propagated to the datacenters. For fault tolerance purposes, committed transactions are only visible after they have been seen by K datacenters.

In our version of SwiftCloud, datacenters store full replicas as in the original implementation. Nevertheless, clients only cache partial replicas. Having full replicas coexisting with the partial replicas considerably simplifies the management of the latter. This poses several advantages in comparison to an ad-hoc architecture where no full replicas, namely authorities, are assumed. Firstly, clients can discard their (partial) replicas at will as long as their updates have been reliably sent to an authority. Secondly, a client can request any fraction from an authority in order to either get a new partial replica, or to expand its own *shard set*. Notice that having an authority also simplifies the integration of state-dependent *shard queries* in the system, very difficult and costly otherwise. Finally, the authority could store which particles each partial replica has in his *shard set*. Thus, it could only propagate operations to the interested replicas, saving bandwidth.

B. Experimental setup

SwiftLinks was evaluated using three Amazon EC2 servers as datacenters: one in Ireland and two in the USA (east and west coast). The EC2 instances are equivalent to a single core 64-bit 2.8 GHz Intel Xeon virtual processor (4 ECUs) with 7.5 GB of RAM. The clients run in 15 PlanetLab nodes located near the DCs. These nodes have heterogeneous configurations with varying processing power and RAM. We set up five SwiftLinks users running concurrently per node, a total of 75. Each client performs an operation per second.

Throughout the evaluation, we use three different modes:

- *Cloud*: This mode simulates a typical geo-replicated system. Clients do not cache any data. Operations are applied synchronously at one datacenter and asynchronously replicated to the rest of datacenters.
- *Partial*. This is the mode that integrates the CPRDTs. Thus, clients only fetch and cache parts of the data structures (CRDTs) as needed.
- *Full*. This is the SwiftCloud approach. Clients cache whole CRDTs even when only part of it is needed.

We limit the capacity of the cache in our experiments to simulate memory restrictions on thin clients, such as a mobile phone. Nowadays, a mobile phone can have up to several gigabytes of memory, but it can easily have tens of applications running simultaneously. An application needs to cohabit with many other applications with limited memory. Therefore, we use 64MB as the default size for cache. If the cache size exceeds this limit, the least recently used object is dropped. In this configuration, *full* and *partial*, if the cache contains the required data, the operations are run locally, and asynchronously propagated to the closest datacenter.

The difference between *full* and *partial* is that the latter benefits from the partial replication mechanism described in the paper. This means that objects are fetched in parts as needed, so the cache can hold parts of an object. For instance, a query for the top ten posts of a forum would only replicate those ten posts in clients cache. On the other hand, for the *full* mode, the objects are only fully replicated in clients side, as in SwiftCloud. Therefore, the same top ten posts query would replicate the whole forum.

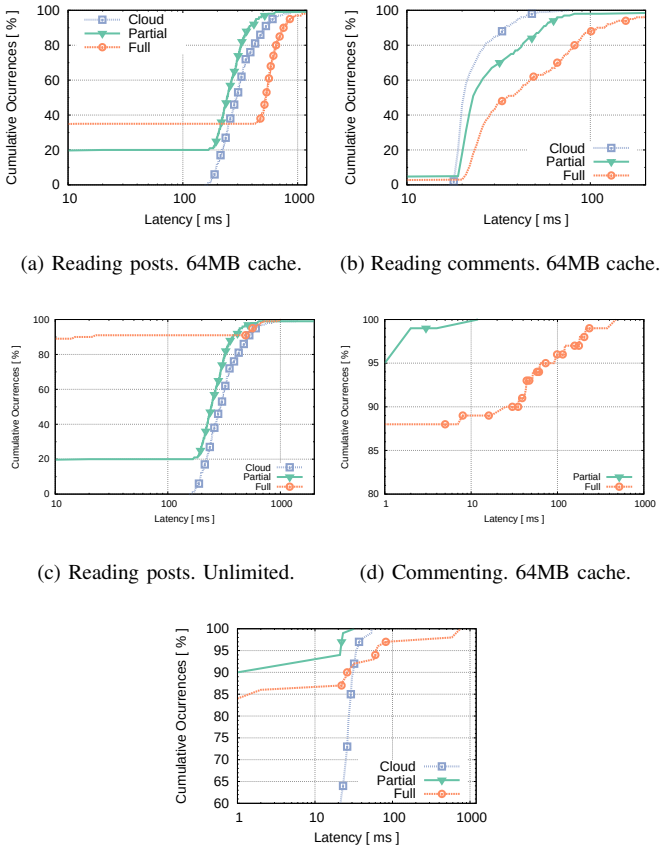
C. Latency

We evaluated the perceived latency for various operations with and without partial object replication. Figure 1 shows the cumulative distribution functions of different operations' latency. These results are obtained after a warm-up phase for the cache. This means that the cache is pre-filled with objects that will be used by the operations present in the workload. For the *full* and *partial* mode, there are always a percentage of operations with a very low latency. We can conclude that it is the percentage of operations that hit the cache.

Read operations Figure 1a shows that the *full* mode has greater cache hit rate (35%) than the *partial* mode. Nevertheless, the hit rate is not optimal due to the limit in the cache size: the cache cannot hold full replicas of all the forums and thus sometimes need to fetch them again. Figure 1c shows the results of a similar experiment but without any cache size limit. In that case, the cache hit rate, for the *full* mode, is 90%, which corresponds to our ratio of biased operations, and it confirms the previous results with a social network application of the SwiftCloud paper [10]. On the other hand, in *partial* mode, the cache hit rate is lower, with only 20% in both experiments (figures 1a and 1c), because the cache only holds partial replicas which gives it less chance of having all the parts needed for hitting the cache in subsequent operations. However, it has the advantage of a lower maximum latency: if an operation does not hit the cache, it only needs to fetch some parts, instead of the full object. In that scenario, it induces a delay similar to the cloud solution, around 200 to 300 ms, while without partial replication, the delay is increased to around 500 to 700 ms by having to replicate a full object. This poses a trade-off between the cache hit rate and the maximum latency. While fully replicating an object will provide more cache hits, a cache miss is more costly.

For the latency of reading comments of a post, shown in Figure 1b, the situation is a bit different. Clients are less likely to read the same comment tree multiple times; therefore, this affects the cache hit ratio. As the figure shows, the hit ratio is less than 5% in both *partial* and *full* replication. But again, *partial* replication has the advantage of reducing the impact of a cache miss as it only replicates the comments required by the operation instead of the full comment tree. In consequence, the *partial* approach has a slightly better latency, close to the *cloud* mode. The *cloud* mode performs better because it does never need to fetch any extra metadata, which means that the returned messages are considerable smaller. Notice that the difference between *full* and *partial* mode has been reduced in this experiment because the involved objects are smaller.

Update operations Caching modes (*full* and *partial*) are more beneficial with update operations. The reason is that update operations are typically applied on objects, or parts of objects, that have already been read by the client. In addition, the update operations only use version-independent queries to fetch their missing parts, which substantially simplifies the comparison of partial objects in the cache. Figure 1e proves experimentally our reasoning. While the cloud mode has an almost constant latency for all operations of a round-trip time, with caching modes, most of the operations (almost 90%) have no latency. Again, the *partial* mode has the advantage of reducing the latency when the cache is not hit, as it only needs to fetch the parts of the object that need to be updated,



(a) Reading posts. 64MB cache.

(b) Reading comments. 64MB cache.

(c) Reading posts. Unlimited.

(d) Commenting. 64MB cache.

(e) Posting, commenting, voting a post, and voting a comment

Fig. 1: CDF of SwiftLinks operation latencies

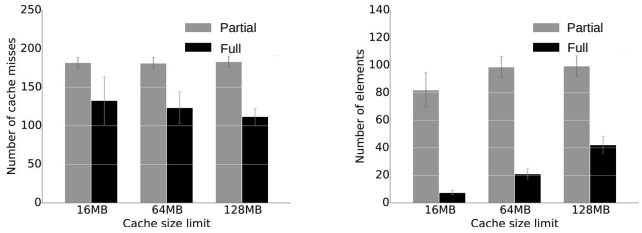
instead of the full object. Moreover, some updates can be done blindly, therefore, they are completed locally.

In particular, Figure 1d shows the benefit of updates when posting comments, which almost always only requires particles already present in the cache. One can see that with partial replication, all the operations have almost no latency, as they can be done completely asynchronously. In contrast, in *full* mode, there can be a large delay when the tree of comments is not in the cache, as it needs to be fetched from the store. As in previous scenarios, even if an operation cannot be done completely locally in *partial* mode, the client only has to fetch part of the tree to complete the update.

D. Impact of cache size limit

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

1) Impact on latency: We have empirically demonstrated that the *partial* mode performs better without cache limit when reading links. We run the same experiments showed in figures 1a, 1b and 1e setting the cache size limit to 16MB and 128MB. The experiments show that a smaller cache (16MB) size limit has a big latency impact on reading links and updates in *full* mode. Nevertheless, its impact is considerable smaller in



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

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

Fig. 2: Impact of cache size limit in partial and full modes

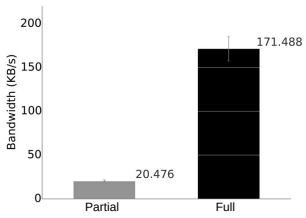
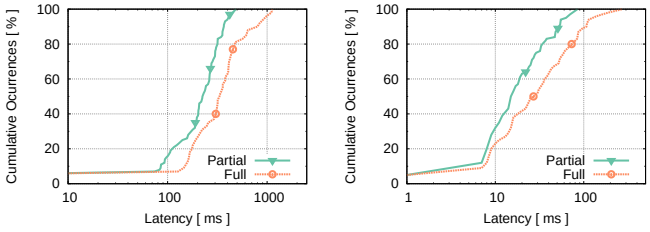


Fig. 3: Average bandwidth usage to fetch objects with a 128MB cache limit, with the cache already warmed up.



(a) Reading posts

(b) Reading comments

Fig. 4: Perceived latency of SwiftLinks during cache warm up.

partial mode. With a small cache, the cache hit rate of *full* mode of reading links becomes worse than in *partial* mode. This is because only a few objects can fit in the cache at a given time; therefore, clients need to fetch objects more frequently. This results in a lower fraction of operations having no latency, about 5% against the 35% obtained with a 64MB cache. There is also an impact for the *partial* mode, but it is considerable lower: it only drops to 13% from 20%. The same applies for update operations. Nevertheless, reads of comments are almost not impacted by the cache size limit: the operations have a low cache locality, so most operations need to fetch an object from the datacenter.

With a 128MB cache size limit, the *full* mode has a large portion of zero latency operations when reading posts, as more are kept in the cache. It however still performs worse than *partial* fetching for operations that do not hit the cache. The latency of updates also improves for the *full* mode with larger cache size, but the *partial* mode still outperforms it.

2) Impact on cache miss rate: The size limit imposed on the cache also has an impact on the cache hit rate. Figure 2a shows that the *partial* mode is less impacted by the cache size limit than the *full* mode. With the three cache limits, the *partial* mode shows a rather stable number of cache misses, about 180. Nevertheless, this does not apply to the *full* mode, where the number of caches misses increases as the cache size is reduced. As in previous experiments, the cache miss rate is greater in the *partial* mode. Nevertheless, we have shown that latency in *partial* mode, is always smaller in average.

3) Impact on number of objects in the cache: The cache size also impacts the number of objects that can be kept in the cache. Notice that for partial replication, only one object is counted even if multiple parts of it have been fetched over time. Figure 2b shows the difference between both modes: *partial* and *full*. In the *partial* mode, many more objects can fit in the cache at any moment, since only parts are kept. 64MB is enough to keep all the objects needed by the application, while in the *full* mode, even 128MB is not enough. This, depending on the workload, may increase the cache hit rate.

E. Bandwidth usage

In *partial mode*, when a client accesses an object, only the needed part of that object is fetched. This can result in saving bandwidth usage compared to *full* mode. In this experiment, we compare the bandwidth usage of *partial* mode and *full* mode. We measure the average bandwidth usage of one client for both over one minute, with the cache already warmed up. Figure 3 shows that the *partial* mode uses only about 12% of bandwidth compared to the *full* mode.

F. Cache warm up

The following experiments compare both *partial* and *full* modes latencies when the cache is still cold, i.e. no objects are stored in the client side. Figure 4 shows the latency of operations during the first 10 seconds of running the application, with a cold cache. In this case, the *partial* mode produces lower latencies as it does not need to replicate the full object. The difference is more noticeable for post reading operations, as shown in Figure 4a, as the set of posts (forums) are large objects. But even for smaller objects, such as comment trees, the *partial* mode outperforms the *full* one (Figure 4b). Notice that the cache size limit does not impact these experiments, since after 10 seconds, the cache does not get full.

G. Discussion

We have seen that partial replication has advantages over full replication of objects. First, it sets an upper bound on the latency of operations by limiting the amount of data that is fetched from the store. Plus, blind update operations gain the additional benefit of being applied locally even if the object is not cached. Second, the cache is more efficiently used, which allows more objects to be kept locally even with a small cache size limit. This is useful for memory-thin devices, and to work on very large data structures with a low memory usage. Third, partial replication also reduces the bandwidth usage of the application by a factor of 8, which is especially valuable on mobile wireless connections, such as EDGE and 3G. Finally, the last advantage is a lower cost of filling the cache when

starting the application. When the cache is empty all operations induce a cache miss, which is especially costly if a large object has to be fetched. Partial replication limits this issue by only replicating the parts of the object that are actually needed.

Unfortunately, *partial* mode limits the cache hit rate, as objects are not fully replicated right away, and non-replicated parts may be needed by subsequent operations. Thus, its use may depend on the workload and the cost of a cache miss. However, a tradeoff is possible between the two: instead of only fetching the parts needed by the operations, one could fetch extra parts of the object. This would however increase bandwidth and cache size utilisation. Latency could be kept low by asynchronously fetching the additional parts.

V. Related work

PRACTI [19] allows clients to select a subset of objects to replicate. Clients only receive updates on objects of their selected subset. However, clients are forced to keep some metadata about objects that they are not interested. Polyjuz [20] stores objects consisting of a set of fields. Clients can decide which fields of each object to replicate. Each subset of fields is denoted as fidelity level. Clients can select different fidelity levels according to the space or network limitations of the device where the objects are replicated. Polyjuz transparently handles the replication of an object in different fidelity levels. In Cimbiosys [21], objects are grouped into collections. Users can use filter expressions to only replicate objects that satisfy some criteria. For example, a user can group his emails in a collection and choose only to replicate emails from his university in his phone. While in the first two systems, users choose the object or fields to replicate based on their name or type, in Cimbiosys user can define replication criteria based on the value of some properties of objects.

VI. Conclusion and future work

We have introduced and formalized a new set of CRDTs called Conflict-free Partially Replicated Data Types, an extension of CRDTs which allows replicas to hold parts of data structures. Our extensive evaluation has shown that CPRDTs can improve the bandwidth and memory usage of replicas by only replicating parts needed by clients, specially in the presence of large data structures under limited cache sizes. Although cache sizes may be larger in the future, we believe that our reasoning will still apply and future applications will still benefit from the CPRDTs approach. The experimental study has also shown that CPRDTs reduce latency in average in comparison to the full mode. However, CPRDTs have a negative impact on the cache hit rate, which has to be weighted against the upper bound on the latency provided.

We plan to extend this work in several directions. First, partial replication can be used as a security mechanism to avoid replicating sensitive data by restricting access with finely grained rules. We believe it is an interesting way of exploiting CPRDTs. Second, we want to study how predictive caching techniques could still improve bandwidth usage and consequently reduce latency even more.

Acknowledgments: We thank Marek Zawirski for his help integrating CPRDTs into SwiftCloud. This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n^o 609551 and by the Erasmus Mundus Joint Doctorate Programme under Grant Agreement 2012-0030.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[2] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *SOSP'11*. New York, NY, USA: ACM, 2011, pp. 401–416.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[4] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and http chunking in web search," in *Velocity Web Performance and Operations Conference*, June 2009.

[5] C. Jay, M. Glencross, and R. Hubbold, "Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment," *ACM Trans. Comput.-Hum. Interact.*, vol. 14, no. 2, Aug. 2007.

[6] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: a highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, p. 447459, Apr 1990.

[7] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in Bayou, a weakly connected replicated storage system*. ACM, 1995, vol. 29.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," INRIA, Rapport de recherche RR-7506, Jan. 2011.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6976, pp. 386–400.

[10] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça, "Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine," *CoRR*, vol. abs/1310.3107, 2013.

[11] "Amazon S3," http://aws.amazon.com/s3.

[12] "Google cloud storage," http://cloud.google.com/storage.

[13] "Windows Azure," http://www.microsoft.com/windowsazure.

[14] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *MobiSys'09*. Association for Computing Machinery, Inc., June 2009.

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

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

[17] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro, "Incremental stream processing using computational conflict-free replicated data types," in *CloudDP'13*. New York, NY, USA: ACM, 2013, pp. 31–36.

[18] I. Briquemont, "Optimising client-side geo-replication with partially replicated data structures," Master's thesis, ICTEAM Institute, Universit catholique de Louvain, Sep. 2014.

[19] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "Practi replication," in *NSDI'06*. Berkeley, CA, USA: USENIX Association, 2006, pp. 5–5.

[20] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *MobiSys '09*. New York, NY, USA: ACM, 2009, pp. 83–94.

[21] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: A platform for content-based partial replication," in *NSDI'09*. Berkeley, CA, USA: USENIX Association, 2009, pp. 261–276.