UNIVERSITE CATHOLIQUE DE LOUVAIN

ECOLE POLYTECHNIQUE DE LOUVAIN

Optimising Client-side Geo-replication with Partially Replicated Data Structures

Supervisor: Peter Van Roy

Readers: Marc Lobelle

Zhongmiao Li

Thesis submitted for the Master's degree in computer science (120 credits) options: networking and security

by Iwan Briquemont

Louvain-la-Neuve

Academic year 2013-2014

Acknowledgements

Support for the evaluation was provided by SyncFree and PlanetLab. I would like to thank Marek Zawirsky for his support to integrate partial replication into SwiftCloud. Thanks to Manuel Bravo for his useful remarks and help throughout the writing of this master's thesis. Special thanks to Xavier Bellekens who read versions of the text and gave sage advice. Thanks to reddit inc. for their website, releasing their source code, and providing API access to their data. I am grateful for my readers, Professor Marc Lobelle and Zhongmiao Li, for their time. Finally I would like to thank my supervisor, Professor Peter Van Roy, for his comments and engagement.

Abstract

Current mobile and web applications replicate data increasingly at the client-side to reduce latency and to tolerate network issues. Mobile devices have however limited storage and network capabilities. Thus large data structures at the server-side need to be split before being sent to the client and reintegrated when modified, in order to keep a low memory and network usage. Ad-hoc solutions tend to be poorly integrated with server-side storage, and do not have well-defined consistency guarantees.

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

Contents

Chapter 1

Introduction

Designer of large user-oriented distributed applications, such as social networks and mobile applications, adopt many measures to improve the performance of their application on end-user devices. Latency is a major concern as people are very sensitive to it, 50ms already annoys users of interactive applications [8], and Greg Linden at Amazon reported that each 100ms delay increase costs them 1% in sales [11]. Similarly, Google's Marissa Mayer found in an experiment that half a second delay added to their search results incurred a 20% drop in traffic [10]. Accessing to data through the network is the principal contributor to latency. It is almost non-compressible as distance and network equipment induce a fixed delay, and, in most cases, the network is not under control of the application developer. Other sources of latency, such as limited computing resources, can often be lessened in many ways, for example by upgrading the hardware or by optimising the performance of the application.

1.1 Geo-replication

Many cloud infrastructures alleviate this issue by geo-replicating data in multiple Data Centres (DC) to get closer to the client and to improve availability. However, there can still be a significant delay to reach the closest DC. Measurements on Facebook DCs showed that they had round trip time ranging from tens to hundreds of milliseconds, and one operation can require several round trips [25]. Moreover, the Internet connections of clients, especially with wireless, may be erratic and fluctuate in latency, or even disconnect.

1.2 Reducing latency through client-side caching

An additional approach is to cache data on the client-side to reduce communications with the DCs. Caching is hard to do right, especially with regards to the data consistency requirements of the application. For read-only caches, the application has to deal with stale data or provide a mechanism to keep the cache fresh, such as cache invalidation, that

needs to be integrated with the server-side storage system. It gets more complicated if we also want *writes* to be cached, that is applied instantaneously in the cache but committed asynchronously to the store, as a *write* of a user could conflict with an *update* of another user. This means that one of those *writes* would need to be rolled back, which further complicates the caching mechanism.

1.2.1 Conflict-free replicated data types

A Conflict-free Replicated Data Type (CRDT) is a replicated data structure ([20, 19, 3]) that can simplify the maintenance of a cache. It allows *read* and *write* operations to be executed locally on a cache and replicated completely asynchronously to the server store, with the cache being a *replica* of the data. It can also keep the cached data fresh by receiving, again asynchronously, changes from the store. This is achieved without any risk of rollback when an operation is replicated: the operations do not conflict. This comes at a well defined cost in consistency: these data types are only *eventually consistent* [7], meaning that different replicas (the caches in our example) might have different states due to seeing the write operations in different orders. This is a weaker guarantee than in usual systems (such as traditional databases), which ensure *strong consistency* but at the cost of requiring *synchronisation* [6]. This stronger guarantee is usually not required for most operations of an interactive application, moreover, CRDTs specify exactly how these differences in state are perceived, to avoid displaying abnormalities to the users. The inconsistencies can be further diminished by providing *causal consistency*, which restricts differences in ordering of operations at different replicas to *causal orderings*, so only *concurrent* operations can be seen in different orders. Chapter 2 explains CRDTs further and show how they can be specified.

To be useful in a cloud infrastructure, these CRDTs need to be integrated at both the server-side store and the client application. We consider here key-value stores. Some solutions already use similar eventually consistent replication mechanisms at the DCs, which allow to simplify the replication and to improve performance by not requiring synchronisation [9, 13, 12, 2, 21], thus reducing the delay for data access and update, but these do not extend to the clients. SwiftCloud is the first system to geo-replicate data all the way to the clients through conflict-free data types [26]. It also provides tolerance to network partitioning and DC failures, and supports transactions with causal consistency guarantees. Chapter 3 describes the architecture of SwiftCloud and briefly explains how it achieves eventual consistency and fault-tolerance.

1.3 SwiftLinks

SwiftCloud has already been evaluated on a social network application to show the gain of client side caching. We built a link-sharing application called SwiftLinks, based on

reddit™, to evaluate SwiftCloud with an application that uses large data structures. Chapter 4 describes the application and the data structures required, along with the corresponding CRDTs that can be used.

1.4 Partially replicated data types

With SwiftCloud, clients hold replicas of the data themselves. Of course, replicating all the objects of an application at the client is not feasible for a few reasons; the amount of data of a large distributed application is simply too big to keep in clients, which do not have the storage capabilities of a datacentre. Also, the traffic generated to keep these objects up to date and consistent is high, and again a client cannot be expected to have enough bandwidth to handle this traffic. For this reason, SwiftCloud clients only replicates objects as they are needed by the application, and can discard a replica when it is not used anymore or when the cache is full. However, an application might use large objects which are hard to split, such as the set of links in the SwiftLinks application. This means that even by only replicating objects as needed, the replicas could still require a lot of storage which would quickly fill the cache, and induce a lot of traffic. To alleviate this issue, we propose to allow single CRDT objects to be replicated only partially. The client replica only holds the part of the data structure it needs. In this way, large objects can be kept as is in the datacentres but only parts of that object need to be replicated at the client, such as a single page of links in SwiftLinks.

Furthermore, the client always has access to the *hollow replica* of an object to do *blind updates*. The hollow replica does not contain any state, but it can ask to apply an update blindly at the DC without any prior communication. Blind updates can thus be done at the client-side without any delay.

We call the replicated structures that support this method of replication Conflict-free Partially Replicated Data Types, chapter 5 shows how they can be built while keeping the same properties as usual CRDTs, and then we describe how they were integrated into SwiftCloud.

1.4.1 Evaluation

As described in chapter 6 where we evaluate our solution with SwiftLinks, bandwidth, memory, and latency usage with partial replication is reduced compared to traditional DC-based geo-replication and to SwiftCloud's caching solution.

1.5 Contributions

We define the concept of partial object replication for CRDTs and show how to keep eventual consistency guarantees. We provide specifications of partial CRDTs for different

data types. We give approaches to implement them in practice, and describe how we extended SwiftCloud to support partial replication.

We build an application called SwiftLinks, based on reddit, with SwiftCloud. We evaluate our SwiftCloud extension experimentally against the existing client-side replication mechanism and usual cloud-based replication.

Chapter 2

Conflict-free Replicated Data Types

Large distributed systems, such as peer-to-peer or cloud computing platforms, require data replication to make them scalable and efficient. Indeed, having multiple replicas of the data can improve failure tolerance, if some replica crashes or becomes unavailable, the system can continue working with the others. Moreover, in a truly scalable system, it should be possible to add replicas to improve its performance. These replicas can also be seen as read-write caches which bring the data closer to where it is needed. We refer to a replica as both the actual data replicated and the logical node the data is kept on, which is connected to the network, when there is no ambiguity between the two.

The replication also introduces a need for a form of data consistency, to allow reading from and updating on any replica while keeping guarantees on what is read. The usual approach is to have *strong consistency* which enforces a serial global order on the updates we want to apply on data, so all replicas see the same sequence of data changes, but not at exactly the same time due to the network delay between replicas. This is however a strong requirement which conflicts with availability of the system and its tolerance to network partitioning according to the CAP (Consistency, Availability, Partition tolerance) theorem [5]. A system is said available if the data can always be accessed in a timely manner, even in case of crashes. Partition tolerance refers to the resistance of the system to failures in the network, which isolates a replica, or a set of replicas, from the others. Those three properties cannot be satisfied at the same time, so one has to sacrifice either availability or partition tolerance for strong consistency. This requirement also limits performance and scalability of distributed systems due to the cost of maintaining strong consistency [24].

2.1 Eventual consistency

Most applications do not require strong consistency for most of their operations, and can use different techniques for these operations. Weakening the consistency guarantees to

eventual consistency [7] alleviates the previous issues. Eventual consistency means that the correct (i.e.: that have not failed) replicas of a shared object will eventually converge to a common state, even though some intermediate states might diverge to a certain extent.

For all (correct) replicas to converge, they must be able to eventually communicate so they can share updates on the data, so network partitioning cannot be permanent to achieve eventual consistency. It is thus assumed that network partitioning is not permanent, which is usually fair to assume. Despite this, we would like for each partition to be at least eventually consistent with itself. This is achieved through *strong eventual consistency*, which requires another condition: all correct replicas that have delivered the same updates have an equivalent state. Thus, a partition will also converge to a common state, even if it does not converge with another partition.

2.2 CRDT

A Conflict-free Replicated Data Type (CRDT) is a concurrent data structure which guarantees strong eventual consistency through simple sufficient conditions [20, 19, 3]. Operations can be applied to a CRDT, query operations to read its state and update operations to modify its state. Which operations are available depends on the data type, for example a set can have a *lookup* query to verify if an element is in the set, a *add* update to add an element, and a *remove* update to remove one.

It does not require any form of synchronisation, so operations are always available, and even if the network partitions, each partition will be eventually consistent with itself, and all replicas will eventually become consistent after the network recovers. Updates can be applied locally and replicated asynchronously without any risk of conflict with other updates. CRDTs can also be designed in ways to limit which inconsistencies can be seen at replicas to better fit the underlying application requirements. Various CRDTs exist such as registers, sets, maps and graphs.

2.3 System model

The distributed system we consider for CRDTs is based on [20] and composed of a fixed, unbounded in size, set of distributed processes. Each process holds one replica on which it can do operations. For simplicity we assume that a replica is only a single object, that is a single instance of a CRDT, e.g. a set, a register, etc. Processes can communicate over an asynchronous network which can partition for a finite period of time. We consider a crash-recovery failure model, where a process can either crash and never recover, or crash and recover with its memory intact.

2.4 Replication

The aim of CRDTs is to have its replicas converge to a common state, and there are two ways to achieve this; the first is operation-based, where *update operations* have properties which allow them to be reordered, so that the operations made at one replica simply need to be sent reliably to the others to achieve convergence. In that case CRDT stands for Commutative Replicated Data Type (CmRDT). The other is state-based, instead of sending operations, the state of a replica is sent unreliably to the others which merge the incoming state with their own in a certain way to converge, this approach applies to Convergent Replicated Data Types (CvRDTs).

In the following sections we give the intuition for these two approaches through examples.

2.4.1 Specification

We can specify how a CRDT works by defining its internal state, called the payload; its *query* and *update* operations; and, for state-based CRDTs, how to merge the payload of two replica so they reach a common state. The local operations are executed atomically: there is no concurrency at a single replica. The way we represent these specifications, as Shapiro et al [20], is quickly described in this section.

Operation-based

Specification 1 shows a model on how to specify an *operation-based* CRDT. First the type of the payload is defined along with its initial value. Then the operations are described, which is the interface of the CRDT. The query operations read the payload and return a value corresponding to the external representation of the CRDT. The update operations are divided into two phases: a prepare phase and an effect phase. The prepare phase is done at the replica of the process doing the update, it does not modify the payload, it creates a value that must be broadcasted to all replicas to apply the update. The effect phase describes how the prepared update must be applied at a downstream replica, that is a replica that receives a prepared update. This phase can modify the payload of the replica.

State-based

State-based CRDTs have a slightly different model shown in Specification 2. The payload and query operations definitions are the same as before. Update operations are simpler: as replication is done through the state of the CRDT, updates are only applied on the local payload. However, there is an additional method to merge the payload of two replicas, which allows them to converge. Note that when we say two replicas converge to a same state, we mean the externally visible state that is seen through the query operations, not

Specification 1 Op-based specification model

```
1: payload type
2:     initial Initial value
3: query query(arguments) : returns
4:     pre Precondition
5:     let Evaluate synchronously, no side effects on the payload
6: update Global update(arguments) : returns
7:     prepare (arguments) : intermediate value(s) to pass downstream
8:         pre Precondition
9:         let 1st phase: synchronous, at source replica, no side effects
10:    effect (arguments passed downstream)
11:        pre Precondition against downstream state
12:        let 2nd phase: asynchronous, exactly once delivery, side effects to downstream
    state
```

the payload which is an internal state. Usually the internal state will also converge but it needs not be the case.

Specification 2 State-based object specification

```
1: payload type
2:     initial Initial value
3: query query(arguments) : returns
4:     pre Precondition
5:     let Evaluate synchronously, no side effects
6: update update(arguments) : returns
7:     pre Precondition
8:     let Evaluate at source, synchronously
9: merge (value1, value2) : payload mergedValues
10:    Least Upper Bound merge of value1 and value2
```

2.4.2 Operation-based replication

Counter

A simple example of a CRDT is a counter. A counter has two operations: reading its value, which starts at 0, and incrementing it by 1. The internal state of the CmRDT is simply the value of the counter, and starts at 0. The read operation returns this value. The replication with operations is trivial: an increment operation made at one replica is reliably broadcasted (i.e. each replica receives the operation once and only once possibly with reordering), and when a replica receives it, it increments its internal value. This

is described in Specification 3. Note that reliable broadcast is possible in asynchronous networks, so it fits in the system model.

This is indeed a CmRDT because each increment operation is reliably broadcasted, so it will be received exactly once by each replica, and more importantly this operation is *commutative*, so regardless of the order the operations are received, all replicas will eventually converge to the same value when no new operations are issued.

Specification 3 Counter, operation-based

```
1: payload ℕ0 count
2:     initial 0
3: query value() : ℕ0 value
4:     let value = count
5: update increment()
6:     effect ()
7:         let count := count + 1
```

Set

Counters are very simple, but it is possible to build much more complex data structures such as sets. A set data structure is a set of immutable elements (by default an empty set) on which we can *add* an element, *remove* an element, and *lookup* whether an element is in the set or not. The *add* and *remove* operations are not commutative as such: if a *remove* is delivered before an *add* on the same element at one replica but the order is reversed at another, the replicas may never converge. An example of such problematic execution is shown in figure 2.1. As you can see, even though the three operations are received by both processes, one process ends up with an empty set while another sees the set $\{e\}$.

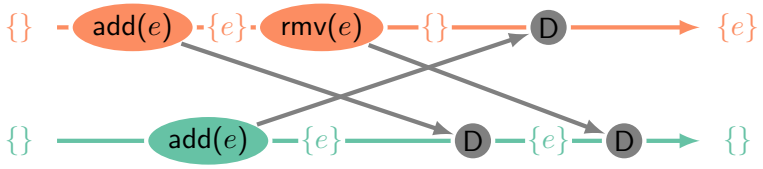


Figure 2.1: Naive op-based set counter-example.

So a CmRDT cannot be built in the same manner as for the counter, additional internal metadata must be kept to make those commutative. A simple way to do this is to keep the removed elements in a separate set. So the state of the CRDT composed of an *add set* and a *remove set*. The received *add operation* adds the element to the *add set* and the *remove operation* adds the element to the *remove set*. A *lookup* on an element at a replica returns true only if the element is in the *add set* but not in the *remove set*. In this way, the operations commute, as we can see on the previously problematic execution in Figure 2.2. This is called a remove-once set, and concurrent *remove* and *add* operations

on an element result in the element not being in the set. However, it also means that once an element is removed, it cannot be added again, which is not what we would expect from a set data type.

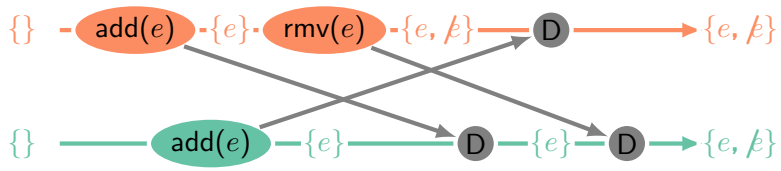


Figure 2.2: Remove-once op-based set correct example.

We can extend on the previous solution to create an Observed-Remove Set (OR-set) which allows for unlimited *adds* and *removes*. The idea is to keep a record of all the *add* operations, even if they are on the same element. And when removing, instead of just adding the element to the remove set, we add all the *add operations* for that element (the *observed adds*) to the *remove set*. An element is in the set if it has at least one *add* that is not also in the *remove set*. An execution example is shown in Figure 2.3. Note that unlike the remove-once set, the *add* operations wins if a remove operation is concurrently done, so even if the execution is the same as in Figure 2.2, the resulting converging state is that e is in the set, instead of the empty set.

In practice, the add and remove sets are sets of pairs $(element, uniqueid)$, where *element* is an element of the set and *uniqueid* is a unique identifier corresponding to a *add*, see Specification 4. So to *add* an element of the set e, a pair (e, id) with a unique identifier id is broadcasted, and when received by a replica, it is added to the *add set*. To remove e from the set, the set of pairs $(e, uniqueid)$ present in the *add set* at the local replica is broadcasted, and when received, it is added to the *remove set*. A *lookup* for an element e at a replica returns true if there exists a pair $(e, uniqueid)$ in the *add set* which is not in the *remove set*.

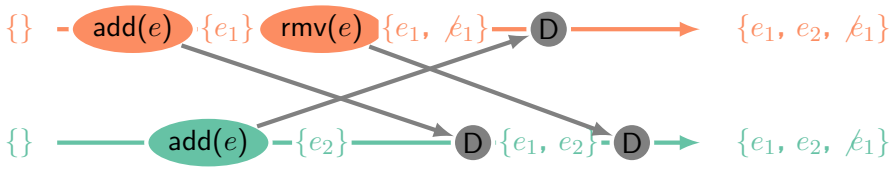


Figure 2.3: Observed-removed set op-based execution example.

Sufficient conditions for convergence

As we have seen on these two examples, for a CRDT to converge, any two update operations must *commute*. Having only commutative operations is not always possible, in particular it is impossible if an operation requires synchronisation. When possible, making the operations commutative can come at a cost in metadata which increases the payload.

Specification 4 Op-based Observed-Remove Set (OR-set)

1: **payload** set A, set R ▷ A = add set of pairs (element, id), R = remove set of pairs (element, id)
2: **initial** $\varnothing$, $\varnothing$
3: **query** lookup(element e) : boolean b
4: **let** $b = \exists u : (e, u) \in (A \setminus R)$
5: **update** add(element e)
6: **prepare** (e) : α
7: **let** $\alpha = unique()$
8: **effect** (e, α)
9: $A := A \cup \{(e, \alpha)\}$
10: **update** remove(element e)
11: **prepare** (e) : set $ToRemove$
12: **pre** lookup(e)
13: **let** $ToRemove = \{(e, u) | \exists u : (e, u) \in A\}$
14: **effect** (set $ToRemove$)
15: $R := R \cup ToRemove$

So far we only depended on reliable broadcast for the delivery of the operations, but with a fixed set of replicas, *causal broadcast* is another possibility in the asynchronous model. With causal broadcast, we restrict the possible orderings of operations to causal orders. In a causal order, only operations which are not causally related, that is which are *concurrent*, can be delivered in an arbitrary order. By using causal broadcast, the requirement of commutativity of operations can be lessened: only *concurrent* update operations must commute, as the causally related operations will be delivered in the same order at all replicas. Sometimes assuming causal delivery allows to simplify the specifications and to remove the need of some metadata in the payload.

For example the OR-set can be simplified by discarding the *remove set*, and instead of adding $(element, uniquedid)$ pairs to the remove set, we can directly remove them from the add set. This works because the $(element, uniqueid)$ pairs are unique, so a pair can only be removed is it has been seen. This means that the *remove operation* of a set of pairs is always causally after the *add operations* of those pairs, thus they do not need to commute.

2.4.3 State-based replication

State-based replication relies on properties of the merge instead of the update operations. The replicas should converge to a common state only through merging with the state of other replicas.

Operation-based and State-based replication equivalence

The previous counter example must be specified in another manner to achieve this. For the counter, we could simply have as payload the set of all increments we have seen (assuming each operation is uniquely tagged, which can easily be done), and the size of this set would be the value of the counter. When receiving the payload of another replica, we can merge it by taking its union with our payload. Eventually, all replicas would have all updates in their payload, and thus would converge to the same state. This is not a very efficient solution but it highlights an interesting result about CRDT replication, we can emulate operation-based replication with state-based and vice versa: they are equivalent [20].

Counter

The actual efficient way to do a state-based counter is by having a payload consisting of multiple counters: one per replica. If we only have one counter, there is no mean to distinguish between two concurrent increments at different replicas. With one counter per replica, the *increment operation* increments the counter assigned to the replica. We assume replicas are numbered from 0 to $n-1$, n being the number of replicas (this is to simplify the specification, the replicas only need unique identifiers). When merging two payloads, we take the maximum of each replica's counter. The value of the main counter is the sum of all counters. This is shown in Specification 5. It is easy to see that this will converge to a common value that corresponds to the number of increments.

Specification 5 State-based counter

1: **payload** $\mathbb{N}_0{}^n$ counters
2: **initial** $(0)^n$
3: **query** value() : $\mathbb{N}_0$ *value*
4: **let** $value = \sum_{i=0}^{n-1} counters[i]$
5: **update** increment()
6: **let** $counters[i] := counters[i] + 1$, with i being the local replica's index
7: **merge** (c, c') : payload *merged*
8: **let** $\forall 0 \leq i < n : merged[i] = max(c[i], c'[i])$

Set

The state-based OR-set is almost the same as the operation-based. Indeed, we can keep the same payload and use the union of the sets as the merge operation, as described in Specification 6, and the replicas will converge to the same sets.

Specification 6 State-based Observed-Remove Set (OR-set)

1: **payload** set A, set R ▷ A = add set of pairs (element, id), R = remove set of pairs (element, id)
2: **initial** $\varnothing$, $\varnothing$
3: **query** lookup(element e) : boolean b
4: **let** $b = \exists u : (e, u) \in (A \setminus R)$
5: **update** add(element e)
6: **let** $\alpha = unique()$
7: $A := A \cup \{(e, \alpha)\}$
8: **update** remove(element e)
9: **pre** lookup(e)
10: **let** $ToRemove = \{(e, u) | \exists u : (e, u) \in A\}$
11: $R := R \cup ToRemove$
12: **merge** (c, c') : payload $merged$
13: **let** $merged.A = c.A \cup c'.A$
14: **let** $merged.R = c.R \cup c'.R$

Sufficient conditions for convergence

As we have seen with operation-based CRDTs and the commutativity of operations, we would like a specific property that ensures a state-based CRDT converges to a common state. We first look at the merge operation. The payloads are sent over an unreliable link, so they might be delivered in incorrect order, dropped, and even duplicated. The merge must make the replicas converge despite this, and we need to find a properties to resolve each of these issues. For the reordering of states, we already have seen a property that solves this with operation-based replication: the merge should be *commutative*. If a payload is dropped, the missing state must be included in subsequent messages to recover, this means that merging has to be *associative*, as thus multiple updates can be combined into the state without loss. Finally, to protect from message duplication, merging should be *idempotent*, so a duplicated message will not change the resulting state after merging. The combination of these three properties means that merge computes the *least upper bound* of a *join semilattice*, which is a *partial order* on the payloads.

Having this property is sufficient to make a CvRDT eventually converge to a common state, but it does not make it correct. More importantly, it might not converge as soon as the same updates have been seen at two replicas, as is required with *strong eventual consistency*. To get strong eventual consistency, the payload of a CvRDT must be *monotonically non-decreasing*, that is an update must make the payload greater than the previous one in the partial order of the join semilattice.

Chapter 3

SwiftCloud

CRDTs provide data types that are synchronisation-free, which can greatly simplify the design of distributed applications. However, we have not shown how to use them to build a practical application. The peer-to-peer system model of the previous section is not well suited to build large scale distributed applications with many clients and lots of objects, such as social networks or online shops. Clients should be able to quickly come and go: they should have access to the data as needed without having to continuously participate in the replication mechanism, and without replicating all the objects of the application. The overhead of keeping strong eventual consistency for clients should be as small as possible, regardless of the number of clients. Also, we have not discussed the fact that real applications would use multiple replicated objects, and there must be some form of consistency between those objects, which calls for support of *transactions* of operations.

SwiftCloud is a cloud storage system which leverages the properties of CRDTs to geo-replicate data to clients while keeping Transactional Causal Consistency guarantees at a small communication overhead [26]. It keeps objects in a key-value store. It uses an operation-based replication mechanism.

3.1 Architecture

All objects are fully replicated at a distributed set of more reliable and more powerful nodes, called *data centres* (DCs). The *client nodes*, also called *scouts*, only replicate objects currently needed by the application as a form of read-write cache. This limits the exchanges with the DCs and improves the responsiveness of the application. The scouts only communicate with the DCs, this allows to track causality between transactions without communicating the causal history of all connected clients. Figure 3.1 shows the general architecture of SwiftCloud.

Each replicated object in the cache or in the DC stores a *prune point*, which is the payload of the CRDT at a specific version, and a log of update operations up to another specific version. This allows the cache and DC to provide the client with versions of the

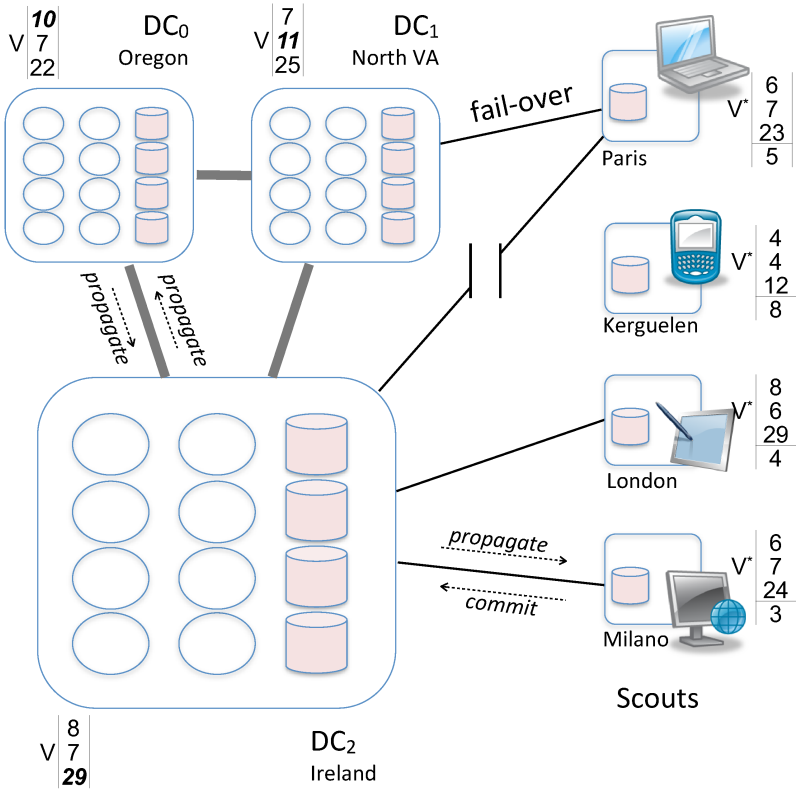


Figure 3.1: SwiftCloud architecture [26]

object that satisfy the consistency guarantees of the transaction, as given in section 3.3. At the same times it allows to reduce the size of objects by moving the prune point to a newer version by applying updates in the log. This is done at the DC when all clients do not need older versions anymore. It also means that SwiftCloud uses operation-based replication. Moreover, it always delivers update operations in *causal order*.

3.2 Application interface

The clients provide an API to the underlying application. The application can execute read and update operations on (possibly multiple) objects within a transaction. The available operations are shown in figure 3.2.

```
begin () : tx_handle
read (tx_handle, object_id) : object
query (tx_handle, object, query_op) : void
update (tx_handle, object, effect_op) : void
commit (tx_handle) : void
rollback (tx_handle) : void
```

Figure 3.2: Client API [26]

`begin`, `commit`, and `rollback` allow to respectively start a new transaction, commit, and roll back an existing transaction. The `read` call fetches the object identified by `object_id`. The `query` and `update` calls allow to do query (`query_op`) and update (`effect_op`) operations on a previously read object. When the application requests an object through a `read`, the scout either has it in its cache and returns it directly, or it fetches the object from the closest DC.

3.3 Transaction guarantees

SwiftCloud includes a transaction model called Transactional Causal+ Consistency, which is weaker than transaction models in traditional databases. It offers *snapshot isolation* but not *serialisability*.

It gives three guarantees:
"every transaction reads a causally consistent snapshot;
updates of a transaction are atomic (all-or-nothing) and isolated (no concurrent transaction observes an intermediate state); and
concurrently committed updates do not conflict" [26].

The first guarantee captures the consistency requirement between objects of the transaction. Once a client sees an update in a transaction, directly or indirectly, it cannot read a version of an object that does not include that update. The state of an object never goes back in time. This is expressed by the concept of a snapshot of the data which includes causally consistent versions of all the objects.

So for example on a social network application with status posts, Alice posts a status A, and Bob, after seeing the status A, posts another status B. If Charlie sees the status B, it will also see the status of Alice.

The second guarantee is mandatory for any distributed system.

The last guarantee follows from the use of CRDTs which are conflict-free.

These transactions can be committed asynchronously, so that clients can commit them first locally, and then commit them to a DC at a later time, when convenient. Unless the client fails completely or is permanently partitioned, all the locally committed transaction will eventually be committed to the store, and cannot rollback.

3.4 Data centre replication

Full replicas are kept in multiple data centres for two main reasons, fault tolerance and performance.

3.4.1 Fault tolerance

First, a client might not be able to reach a data centre because of network partitioning, or the DC has failed, or is in maintenance. The client should be able to fall back to another DC seamlessly, and its waiting transactions should be committed to the new DC. Even with CRDTs, this is not trivial as the causal consistency requirements should still be ensured in case of failure. Depending on the replication mechanism used, there could be different updates seen at each DC. So an update present at one DC might not be available at any other DC, meaning a transaction depending on that update could not be committed globally by a client that is disconnected from that DC. A solution would be to synchronise the DCs so that a consistent snapshot read at one DC is ensured to be present at all, or at least several, DCs. This would unfortunately delay the reads and even possibly undermine the availability of the system.

SwiftCloud solves this by only allowing *safe* reads of the data. A safe read is a read of possibly stale data that only depends on updates that are replicated at multiple DCs. The number of DCs at which an update must be replicated to be considered safe is configurable and is a trade-off between fault tolerance and freshness of the data. Let this number be K. With $K > 1$, if a client reads a snapshot at one DC that later fails or is disconnected, it can then fall back to other DCs until it finds the one that has the same snapshot. This means that the system tolerates up to $K - 1$ DC failures, but an update is only seen once it has been replicated to K DCs.

3.4.2 Performance

Second, having multiple replicas moves the data closer to the clients. Indeed, each client can thus connect to the closest DC to fetch objects and to commit transactions, which reduces the latency of those operations. Moreover, increasing the number of DCs allows to scale the distributed system to handle more clients at the same time.

Chapter 4

SwiftLinks: a vote-based content-sharing application

SwiftLinks, based on reddit^{TM} ([16]), is a link-sharing social network where users can post links, messages, comments, and vote on each other's content. Users can upvote posts and comments that they deem relevant or interesting, and downvote those that they do not. Anyone can then browse the links on topics that interest them, ordered according to their associated votes. The reader is invited to visit `http://www.reddit.com` to interactively see how the application it is based on works.

4.1 Overview

Figure 4.1 shows a page on the subject of programming from the reddit website. The links, or posts, are divided into forums, or subreddits, which gather the community on a specific subject. In the figure you can see at the number 2 that the forum is named `programming` along with other information about the forum. A registered user can publish links to this forum.

At 1 there is a list of forums that the user follows, from which links are picked to appear on the user frontpage. Under 3 there are buttons called *hot*, *new*, etc. which allow to show the links in different orders according to some criteria. For example, the new sorting simply shows the newest links first, while hot favours new links that also have a good score. The score of a link is the difference between the number of upvotes and of downvotes. At the right of 4 you can see the "hottest" link of the forum which has a score of 740. The orange upward arrows shows the user has upvoted that link, while the blue arrow near 5 indicates a downvote. Any registered user can upvote and downvote links. They can also post interesting links to forums, or even create a new forum.

A user can look at the comments made on any link, which leads to a page as in Figure 4.2. The comments are organised as a tree. As for the page of links, those comments can be ordered in different manners and can also be upvoted and downvoted. To publish

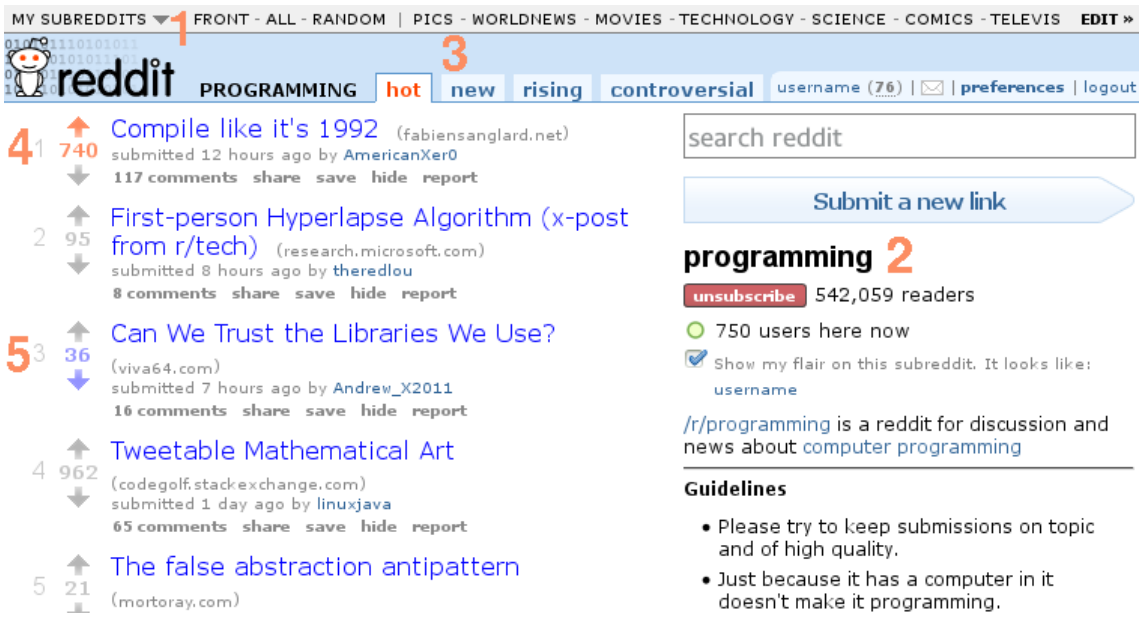


Figure 4.1: Typical page of links from the reddit website.

a new comment, a user can either reply to the link or to another comment.

These two figures (4.1 and 4.2) show the main features of reddit that we implemented in SwiftLinks.

4.2 Why reddit?

This application is well suited for SwiftCloud for multiple reasons.

Firstly, it is interactive, so it should have a low latency, which is one of the main goals of replicating data at or near the client.

Secondly, it is largely a content-driven application: most of its features can be ran on less powerful machines, near or at the client side, with the datacenters focusing on providing and storing content. Also, the content of the actual application represents large amounts of data. This produces a challenge for the replication of the data on mobile devices: those devices have a limited memory size and cannot cache many large objects, and replication should be more fine-grained than the object level to fit the constraints. This challenge is the focus of chapter 5. On the other hand this content gives an opportunity to assess the performance of the clone with real data.

Also, the application has highly dynamic content: it aims to provide fresh and interesting content by sorting it based on the votes of the users and the freshness of the content. But paradoxically, giving stale content to the user for performance reason does not impact the user experience much, being a few tens of seconds behind the content available at the DC is not noticeable, the real reddit website actually uses such an aggressive caching system. Thus SwiftCloud's concept of *reading in the past* to minimise consistency overhead as well as its caching mechanism do not degrade the application usability.

Figure 4.2: Part of the comment tree for the previous link.

It must be highly scalable, because of the large amounts of content created by the users, and to handle its enormous traffic: reddit is in the top 100 visited websites (according to Alexa) [1], and it serves billions of pages per month [16], more than 5 billions at the time of this writing. With eventually consistency, both the data storage and computation system are easier to distribute. Most of the features of reddit do not require synchronisation. In particular, the main features of submitting links, posting comments and voting on content, can be done with conflict-free data structures.

Finally, the code of reddit is open source [17], which gives some insight into its design and how it is optimised.

4.3 Data structure

The data structure of SwiftLinks is described in general terms in the following sections.

4.3.1 Posts

A post is a link posted by an *user* to a specific *forum*. The posts can be upvoted and downvoted by the users to influence their order on the pages.

A post is a record with the `username of poster`, a `publication date`, a `link`, an optional `content text` and the `forum name` where it was posted. Each post is also associated with `vote counter` containing the upvotes and downvotes of the post made by users.

Visitors can view the posts of a forum according to different orderings: by date or by other orderings which are also influenced by the upvotes and downvotes of the posts. Those orderings are described into details in Appendix A. The posts of each forum are kept in an ordered set with multiple orderings.

4.3.2 Comments

A comment is a response to a post or to another comment by a user, and it can also be voted up and down. Thus each post has an associated comment tree that can be read.

Each comment is made of the `username`, a `publication date` and the `comment content`. As for the posts, it also has an associated `vote counter`, and the comment trees can be viewed in multiple orders.

4.3.3 Vote counter

As said, both posts and comments can be voted up and down by users. The vote counter stores, for each user that made a vote, the direction of the vote. The *score* of a post or a comment is the difference between the number of upvotes and the number of downvotes.

4.3.4 Forums

A forum groups posts about a similar topic, it can be created by users.

It is a record with the `forum name` and a `forum description`.

4.3.5 Users

A user is a registered person who can post links, comments, or subscribe to forums.

Its data structure holds a `username`, which is unique, a password, and the various preferences of the user (e.g.: language). It is also associated with a set of forums to which it is subscribed, and only posts from those forums will appear on the frontpage of the user.

4.4 CRDTs

We can translate the given structure into suitable CRDT specifications. These will be implemented within SwiftCloud to build the SwiftLinks application, which we will discuss

further in chapter 6.

4.4.1 Vote counter

The heart of SwiftLinks is its vote system. A user can vote on an object and change his vote subsequently. And it might also make two concurrent votes on the same object, e.g. in two tabs of a browser, or on his computer and on his phone. The votes of different users on a same object do not conflict with each other and thus those operations naturally commute.

We therefore focus on the vote of one user. The last vote made by this user should be the one seen by the clients once they have received the corresponding update. This can be done by considering each vote as a Last-Writer-Win (LWW) register, which is a CRDT. But this does not resolve the issue of concurrent votes by a same user.

There are two ways to deal with this. We can either create a rule to choose a winner in case of concurrent votes, or we keep a trace of those concurrent votes and expose this fact to the application. We describe both in the following sections. We implemented the LWW-based vote counter in SwiftLinks due to its lower memory footprint.

LWW-based vote counter

Our first solution use a LWW register with a specific rule of concurrency conflict resolution. A generic LWW register keeps a monotonically growing timestamp along the value of the register. When we change the value, a new (greater) timestamp is computed (e.g. with a counter), and this change is replicated (either through updates or through state). To merge or to apply an update at a replica, the value with the greatest timestamp is kept (and the greatest timestamp itself). There is of course the possibility that two concurrent updates include the same timestamp, and in that case we must decide between those differently. The simplest way is to take the greatest value according to a total ordering, another possibility is to choose according to the origin of the update.

We apply the first possibility for our vote counter. We give a total order to the 3 votes possible (upvote, no vote, downvote), and take the greatest in case the timestamp are equal.

Last observed vote counter

The previous solution resolves the concurrent votes quite arbitrarily, which may not be the way the user expects it to act. Another possibility would be to show the user there was concurrent votes and allow him to choose again. Also, removing a vote only makes sense if there was a vote before, and the LWW solution does not capture this.

We can build a solution in a way similar to OR-sets (see section 2.4.2). We can record all vote operations uniquely in the payload, and when a vote needs to be changed, we can

look at the vote operations we observed and *cancel* them.

To do this we keep two sets, an upvotes set and a downvotes set, see Specification 7. These sets contains pairs (v, id) where v is a voter and id is a unique identifier. The vote of v is considered middle (no vote) if both sets contain the same pairs by voter v, and is considered up or down if one set (upvote or downvote set) contains all the pairs of the other and at least one more. The last case is if both sets contains pairs that are not in the other, this means that two different votes were made concurrently, so the resulting vote is unknown. Casting a vote is then simple: to cancel a vote we make our pairs (from the voter v) the same in both sets, by taking the union of those pairs and putting them in both sets. To upvote or downvote, we first cancel the existing vote then add another unique pair in the upvote or downvote set, respectively.

This specification can be optimised if we assume causal delivery of the updates. Indeed, instead of copying observed unique votes from one voting set to the other, with causal delivery we can instead remove those observed votes.

4.4.2 Link set

For the set of links, a simple observed removed set, as described in 2.4.2 is be used.

4.4.3 Comment tree

The comments of a link are organised in a specific kind of tree, with the root being the link, as users can respond to other's comments. A tree data structure has some constraints which might create issues with concurrent updates. Specifically if two clients concurrently create the same node but with different parents, those two updates would conflict. We can address this by making the simplifying assumption that the nodes are unique, thus avoiding the conflict. This is an acceptable assumption for our specific application.

Another possibility is to consider each node as not only identified by its value but also by its position in the tree. So a node is defined by its path from the root to the node itself, as a list of the value of each node. We chose this solution in our implementation.

We assume that the root of the tree is a fixed node without value, even if the conceptually the root of the tree is a link, as this does not have any effect on the specification. This data structure could thus be described as a forest instead of a single tree.

The data structure is then specified like a set, with nodes being elements of the set, with one difference: removed nodes should still be visible by the application, but as deleted, to not create holes in the tree. We chose a remove-once tree structure, as comments are not undeleted. It is described in specification 8.

Specification 7 Op-based Last Observed Vote Counter

```
1:  payload set U, set D   ▷ U = observed upvotes (voter, id), D = observed downvotes (voter, id)
2:      initial ∅, ∅
3:  query voteDirection(voter v) : direction d
4:      let ou = {(v, u)|∃u : (v, u) ∈ U}
5:      let od = {(v, u)|∃u : (v, u) ∈ D}
6:      if od ⊂ ou then
7:          let d = up
8:      else
9:          if ou ⊂ od then
10:             let d = down
11:         else
12:             if ou = od then
13:                 let d = middle
14:             else                                   ▷ (ou ∪ od) − (ou ∩ od) ≠ ∅
15:                 let d = unknown
16: update vote(voter v, direction d)
17:     prepare (v) : α, ou, od
18:         let α = unique()
19:         let ou = {(v, u)|∃u : (v, u) ∈ U}          ▷ Observed upvotes to cancel
20:         let od = {(v, u)|∃u : (v, u) ∈ D}          ▷ Observed downvotes to cancel
21:     effect (v, d, α, ou, od)
22:         U := U ∪ od
23:         D := D ∪ ou
24:         if d is up then
25:             U := U ∪ {(v, α)}
26:         if d is down then
27:             D := D ∪ {(v, α)}
```

Specification 8 Operation-based Remove-once Tree.

1: **payload** set A, set R ▷ A = added nodes, R = removed nodes, as tombstones. A node is a pair (parent, value)
2: **initial** $\varnothing$, $\varnothing$
3: **query** root() : node n
4: **let** $n = ()$
5: **query** children(node *parent*) : set *children*
6: **let** $children = \{(p, n) \in A | p = parent\}$
7: **query** isRemoved(node n) : boolean b
8: **let** $b = n \in R$
9: **update** add(node *parent*, value v)
10: **prepare** node *parent*, value v
11: **pre** $parent \in A \wedge parent \notin R$ ▷ The parent node is added and not removed
12: **pre** $(parent, v) \notin (A \cup R)$ ▷ The node to add does not exist and is not already removed
13: **effect** node *parent*, value v
14: **let** $A = A \cup \{(parent, v)\}$
15: **update** remove(node *node*)
16: **prepare** node *parent*, value v
17: **pre** $node \in A \wedge node \notin R$ ▷ The node to remove is added and not yet removed
18: **effect** node *node*
19: **let** $R = R \cup \{node\}$

4.4.4 Forums

Forums can be created by users, and users can subscribe to forums. Each forum can be modelled as a LWW register to put its description. The forum name is unique and can simply be the key of the CRDT instance. Ensuring the uniqueness of a forum should be done synchronously, as it cannot be done with CRDTs. The set of all forum names can be kept in a OR-set.

Similarly, the set of forums a user subscribes to can be modelled as an OR-set.

4.4.5 Users

A username is like a forum name: it must be unique. Its creation should thus be done synchronously. The other pieces of information of a user can be kept in a LWW register. Moreover users are a bit more complicated as they can log in with a password. This security-oriented operation is not supported within SwiftCloud, so logging in synchronously is mandatory.

Chapter 5

Conflict-free Partially Replicated Data Types

Convergent or Commutative Replicated Data Types (CRDTs) ensure *strong eventual consistency* for all the replicas of an object.

For CRDTs each replica contains all the data of the replicated object, even though the operations done at one replica may only involve a fraction of the data. Also, in operation-based replication all the operations done in one replica will need to be sent to all the replicas, and in state-based replication the full state of one replica will need to be shared with all the other when merging. For practical usage of CRDTs, this has two main drawbacks: first a thin client (e.g.: mobile device or embedded computer) might not be able to keep the multiple full replicas, required by usual applications, in memory, second the clients holding a replica will have an overhead in network usage to keep the eventual consistency even for the parts of the data they do not need. Allowing partial replicas also has other applications, a replica could be made partial to enforce fine-grained security restrictions. It also provides a way to support data with multiple fidelity requirements to accommodate resource-thin devices while keeping consistency between the fidelity levels [23], for example by not replicating less important information on mobile devices.

We propose Conflict-free *Partially* Replicated Data Types (CPRDTs) where replicas can be partial while still keeping operation capabilities and being eventually consistent.

This poses many new challenges: all operations are not available on partial replicas, which means new preconditions must be added to ensure correct usage. However these conditions cannot interfere with the convergence of the replicas, with convergence itself being now on partial replicas. Care must be taken as a partial replica may change over time, it could change the parts it keeps, which must be done while keeping eventual consistency and without losing data.

5.1 System model

We consider the same model as in section 2.3, with additions to support partial replicas.

Each process replicates some objects: an object is a named instance of a CRDT. It exposes an interface to clients to read, using query operations, and update, with update operations, the object. What can be read through the interface is called the *data* of the object, and *metadata* is the additional information kept internally to ensure eventual consistency. An update operation can have preconditions that capture its safety requirements, and an operation is said *enabled* at a replica if its precondition is satisfied at that replica. For example, for a set object with a remove operation, the precondition to remove an element is that the element is present in the set.

We further consider that a process might replicate an object partially: it only has access to a part of the data that is of interest to the client and the process only keeps the metadata required for the given part. We assume that for each object the union of all replicas of the processes make up a full object. Partial replication means that data is not automatically replicated at all processes, only the overlapping parts between processes have multiple replicas. This must be taken into account as the redundancy of data is lessened by having partial replicas.

We assume that the part of an object held by a process does not change over time. We will relax this assumption in section 5.6.4.

5.2 Definition

An object can be partially replicated at a process. Intuitively, this means only some part of the data structure is replicated: some elements of a set, a subgraph of a graph, or a slice of a sequence. The part replicated is defined on the data structure of the object.

A CRDT can be divided into *particles* which are the smallest meaningful elements of the *data structure* applicable for query and update operations on the CRDT. On simple set CRDTs, a particle would be any element that can be added to the set, or looked up. Thus the set of particles of a CRDT can be infinite, and we call π the set of all particles of a CRDT. No particle depends on the state of the object. What is the smallest meaningful element depends on what operations are available on the CRDT. A more formal definition of particle is given in Appendix B.

A replica has a state, which is the externally visible value of that replica, and this state can be divided into smaller states associated to each particles. On the set example, the state associated to a candidate element of the set (which is a particle), is whether that element is in the set or not. Each particle can be mapped to internal data and metadata of the CRDT which fully define the associated state of that particle. On the OR-set described in section 2.4.2, a particle would be an element e, and the data and metadata associated to this particle would be the (e, u) pairs of both the add and the remove sets.

Each replica of an object is associated with a set of particles which is the part this replica knows the state of, that is the part on which it can do query and update operations, and also receive update operations of other replicas. For a replica x_i, this set is shard(x_i), which we call shard in reference to database shards. If shard(x_i) = π then x_i is a *full* replica and is equivalent to a normal CRDT. Another specific value is if shard(x_i) = $\varnothing$, then x_i is a *hollow* replica (as named in [15]), it does not hold any state but it can still produce updates, as explained in section 5.3.3. It is assumed that this set is chosen when creating the replica and is not modified afterwards.

We can associate the operations available on a CRDT with the particles the operation needs to know the state of to execute. For an operation with its arguments, *op*, required(*op*) is the set of particles needed by *op*. This means that, for replica x_i, an operation is available only if required(*op*) $\subseteq$ shard(x_i) (all the needed particles for the operations are available in the replica x_i) is satisfied, otherwise the operation does not have enough information to be completed.

For the lookup operation of a set, required($lookup(e)$) = e where e is an element of the set, because we need to know the state of the particle e to know whether e is in the set or not.

Similarly, an update operation can be associated with the particles that might have their state affected by it. We define the affected(*op*) function, which gives the set of particles on which the operation with its arguments *op* might have an effect (the state of these particles could be changed).

5.3 CPRDT replication

There are two equivalent ways to make CRDTs replication converge. The first one is state-based, the state of the CRDT at each replicas is sent to the others to be *merged*. The other is operation-based, operations prepared at one replica are broadcasted to be applied at all replicas.

We already have CRDTs which are strongly eventually consistent objects for full replicas, but we now must consider that replicas may be partial, thus the consistency guarantees can only be defined on the common parts of any two replicas.

Let's first define the state equivalence of two partial replicas: x_i and x_j have equivalent common abstract states if all `query` operations q, for which required(q) $\subseteq$ (shard(x_i) $\cap$ shard(x_j)), return the same values. The strong eventual consistency of CPRDTs is the same as for CRDTs but the convergence property is on the particles shared by any two replicas. Also, the updates that must be seen by a replica are only those that affect the shard of the replica. We define this more formally in the next section.

5.3.1 Causal history and convergence

One requirement for replicas to converge is that they apply, directly or indirectly, the same update operations. We can define what updates operations have been applied as the causal history of a replica. The definitions differ whether one uses state-based or operation-based replication. Indeed, state-based replication both directly apply operations when doing a local update and indirectly when merging their state with a remote replica. While operation-based replication only has updates that are applied when received.

The *merge* method used by a replica must only merge the state of its particles with the remote replica and ignore the others, so that $\text{shard}(x_i) = \text{shard}(x_i \bullet \text{merge}(x_j))$. We define the causal history of a replica for state-based replication as follows:

Definition 1 (Causal History on Partial Replicas - state-based)**.** *For any replica x_i of x:*

- *Initially, $C(x_i) = \varnothing$.*
- *After executing* `update` *operation f,*
  if $\text{affected}(f) \cap \text{shard}(x_i) \neq \varnothing$ *then* $C(f(x_i)) = C(x_i) \cup \{f\}$,
  otherwise $C(f(x_i)) = C(x_i)$.
- *After executing merge against states x_i, x_j,* $C(x_i \bullet \text{merge}(x_j)) = C(x_i) \cup \{f \in C(x_j) | \, \text{affected}(f) \cap \text{shard}(x_i) \neq \varnothing$

Only the updates that affect the replica are included in its causal history, also when merging.

We do the same for op-based replication. The update operations are divided into two phases. The preparation phase which is done at the source replica and does not have any side-effect (it can be seen as a query operation), it only prepares data for the downstream phase. This second phase is applied at all replicas and it affects the state of the replica.

Definition 2 (Causal History on Partial Replicas - op-based)**.** *For any replica x_i of x:*

- *Initially, $C(x_i) = \varnothing$.*
- *After executing the* `downstream` *phase of operation f at replica x_i,*
  if $\text{affected}(f) \cap \text{shard}(x_i) \neq \varnothing$ *then* $C(f(x_i)) = C(x_i) \cup \{f\}$,
  otherwise $C(f(x_i)) = C(x_i)$.

Now we are ready to define convergence:

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: $\forall i, j : C(x_i) = C(x_j)$ implies that the abstract states of i and j are equivalent on their common particles.*

- *Liveness:* $\forall i, j : f \in C(x_i)$ *implies that, eventually, if* $\text{affected}(f) \cap \text{shard}(x_j) \neq \varnothing$, *then* $f \in C(x_j)$.

5.3.2 State-based partial replication

In state-based replication, update operations modify the local state of the replica. This state, or payload, is regularly sent to other replicas which *merge* it with his own. Thus, all updates are eventually seen by all the replicas, directly or indirectly.

This style of replication is interesting if the size of the state is relatively small compared to the size and number of updates, as only the state must be sent over the network. Having partial replicas can help since only parts of the state need to be sent and received.

To achieve convergence with state-based replication on partial replicas, an additional condition is needed for an update to be available at a replica. Since updates are indirectly replicated though the state, an operation cannot be applied if it affects a particle that is not in that replica's shard, 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 $\text{affected}(f) \nsubseteq \text{shard}(x_j)$.

Since the replicas only converge on their common parts, a replica x_i just needs to send to another, x_j, the state of the particles $\text{shard}(x_i) \cap \text{shard}(x_j)$ for them to converge.

5.3.3 Operation-based partial replication

The second way to do replication is by sending updates instead of the states. This means that there is no merge method, the merging is implicit.

With full replicas, the prepare phase of an update is done locally, and its result is broadcasted reliably (sometimes in causal order) to all the replicas which apply the downstream part of the update on their state.

With partial replicas, the difference is that an update is broadcasted only to the replicas that are affected by that update. So an update u is broadcasted to x_i if $\text{affected}(u) \cap \text{shard}(x_i) \neq \varnothing$. It follows that certain updates can be prepared at a replica but not applied at that same replica. These are called *blind updates*, they are not possible with state-based replication. A hollow replica, which has a empty shard, can only do blind updates.

Also, the downstream phase of an update cannot have any required particles, as otherwise if the downstream phase has some required particle r and one different affected particle a, it would not be applied at a replica x_i for which $r \notin \text{shard}(x_i)$ and $a \in \text{shard}(x_i)$ even though it should be applied to converge.

5.4 Specification

We extend the specification models given in section 2.4.1 to describe the required and affected particles of the operations of a CRDT.

5.4.1 Creation of a new partial replica

Although we assumed that the set of replicas as well as the shards of the replicas are fixed, in practice for CRDTs to be useful we need a way to create a new replica by copying an existing one. Without partial replication, the CRDT can simply be copied in its entirety, but for CPRDTs we would like to choose which particles we copy. We achieve this by adding a special operation to create a new replica from a subset of a CPRDT. The subset we want to copy in the new replica is defined by a set of particles. A set of particle is the canonical form to define the subset, in practice it can be defined in a using criteria which depend on the current state of the replica, for example "give me the first 10 elements of your sorted set", which can then be transformed into a set of particles, we discuss this further in section 5.6.1.

We call this operation fraction and for $x_j = \text{fraction}(x_i, Z)$, where Z is a set of particles we want to take, we have $\text{shard}(x_j) = \text{shard}(x_i) \cap Z$ and x_j only contains the data and metadata needed for the particles in $\text{shard}(x_j)$.

This operation is also useful to simplify the specifications of state-based CRDTs: when merging two partial states, we only want to merge the state of the common particles, the rest can be ignored. However, putting this in the specification is cumbersome, as the merge operation must be aware of the shard of the payloads it needs to merge. Instead, we assume that the merge operation merges the complete payloads, regardless of their shard. We can then limit the growth of the replica to its own shard as such: if we are replica x_j, receiving the payload of replica x_i, we do $x_k = \text{fraction}(\text{merge}(x_i, x_j), \text{shard}(x_j))$, thus $\text{shard}(x_k) = \text{shard}(x_j)$ and the replica does not grow. In practice the fraction operation can be applied before sending the payload to another replica (to save bandwidth and computation), but to keep the convergence property, the fraction taken from replicas x_i and sent to x_j must have at least the particles $\text{shard}(x_i) \cap \text{shard}(x_j)$, otherwise some changes in the state made at x_i might not be received by x_j.

5.4.2 Specification model

The specifications are similar to CRDT specifications, with some added notations. Each operation must define which particles it involves (*required particles* and *affected particles*). Note that the conditions given in section 5.2 ($\text{required}(op) \subseteq \text{shard}(x_i)$, and $\text{affected}(op) \subseteq \text{shard}(op(x_i))$ for state-based replication), regarding whether an operation is enabled or not, are not explicitly given but must be enforced by the system. Also, the specification hides the fact that we might be working on a partial replica: the function shard is not

needed. Everything concerning it can be done through the *fraction* function.

Specification 9 gives the template for state-based CPRDTs, Specification 10 gives it for operation-based objects.

Specification 9 State-based object specification with Partial Replication

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

5.5 CPRDT examples

In this section we give the specifications for some CPRDTs, based on CRDT specifications by Shapiro et al ([20, 19]), except for the tree CPRDT.

5.5.1 Sets

Grow-Only set

Specification 11 gives a simple state-based grow only set (which only support the add operation).

Observed-Removed set

Specifications 12 and 13 show an Observed-Removed set with partial replication capabilities. The operation-based specification assumes causal delivery of its operations to

Specification 10 Op-based specification model with Partial Replication

particle definition Informal definition of what is a particle
query *query(arguments)* : *returns*
    required particles *Set of required particles* ▷ = required(query(arguments))
    pre *Precondition*
    let *Evaluate synchronously, no side effects*
update *Global update(arguments)* : returns
    prepare (*arguments*) : *intermediate value(s) to pass downstream*
        required particles *Set of required particles to prepare the update* ▷ = required(Global update$_{prepare}(arguments)$)
        pre *Precondition*
        let *1st phase: synchronous, at source, no side effects*
    effect (*arguments passed downstream*)
        required particles *Set of required particles when applying the update* ▷ = required(Global update$_{effect}(arguments)$)
        affected particles *Set of particles which might be affected when applying the update* ▷ = affected(Global update$_{effect}(arguments)$)
        pre *Precondition against downstream state*
        let *2nd phase: asynchronous, side effects to downstream state*
fraction (particles *selection*) : payload *partialReplica*
    Copies the particles selection into partialReplica so that shard(*partialReplica*) = *selection* ∩ shard(*self*) *(self is the replica on which fraction is applied to).*

Specification 11 State-based Grow-Only Set (G-set) with Partial Replication

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

optimise the payload. A particle is defined as an element of the set.

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

Specification 12 Op-based Observed-Remove Set (OR-set) with Partial Replication

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

5.5.2 Trees

Grow-only tree

A state-based grow-only tree is specified in Specification 14. A node is defined by its path and its content in a recursive way, which is noted $(parent, nodeContent)$ where $parent$ is

Specification 13 State-based Observed-Remove Set (OR-set) with Partial Replication

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

defined similarly, and the root is empty: (), so for example we can have $(((), 1), 2)$ which is a node with content 2 and with parent $((), 1)$. This allows to make a grow-only tree that is very similar to a set, with only the added precondition that the parent must exist when adding a node. It also means that adding nodes which have the same value and the same parent result in one node in the tree.

The particles for this tree is the nodes (with their parent, as defined).

Specification 14 State-based Grow-Only Tree (G-tree) with Partial Replication.

1: **particle definition** A node of the tree.
2: **payload** set A
3: **initial** $\varnothing$
4: **query** lookup(node n) : boolean b
5: **required particles** $\{n\}$
6: **let** $b = n \in A$
7: **update** add(node $(parent, content)$)
8: **required particles** $\{parent\}$ (if parent is not the root)
9: **affected particles** $\{(parent, content)\}$
10: **pre** $parent \in A$
11: $A := A \cup \{(parent, content)\}$
12: **merge** (S, T) : payload U
13: **let** $U.A = S.A \cup T.A$
14: **fraction** (particles Z) : payload D
15: **let** $D.A = A \cap Z\}$

5.5.3 Directed Graph

In this example, in Specification 15, a particle of the graph is defined as a vertex combined with its outgoing edges: so if vertex v is in the set of particles of the replica, then $\forall v' \in \pi$: edge (v, v') are implicitly there too.

5.6 Practical usage

5.6.1 Shard definition

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

Specification 15 Op-based Directed Graph with Partial Replication

particle definition A vertex with its outgoing edges.
payload set V, E ▷ Sets of pairs {element e, unique-tag w), ...}
    initial $\varnothing, \varnothing$ ▷ V: vertices, E: edges
query lookup(vertex v) : boolean b
    required particles {vertex v}
    let $b = \exists w : (v, w) \in V$
query lookup(edge (v', v'')) : boolean b
    required particles {vertex v', vertex v''}
    let $b = lookup(v') \wedge lookup(v'') \wedge (\exists w : ((v', v''), w) \in E)$
update addVertex(vertex v)
    prepare (v) : w
        let $w = unique()$
    effect (v, w)
        affected particles {vertex v}
        $V := V \cup \{v, w\}$
update removeVertex(vertex v)
    prepare (v) : R
        required particles {vertex v}
        pre $lookup(v) \wedge \nexists v' : lookup((v, v'))$
        let $R = \{(v, w) | \exists w : (v, w) \in V\}$
    effect (R)
        affected particles {vertex v}
        $V := V \setminus R$
update addEdge(vertex v', vertex v'')
    prepare (v', v'') : w
        required particles $\{v'\}$
        pre $lookup(v')$
        let $w = unique()$
    effect (v', w'', w)
        affected particles $\{(v', v'')\}$
        $E := E \cup \{((v', v''), w)\}$
update removeEdge(vertex v', vertex v'')
    prepare (v', v'') : R
        required particles {vertex v', vertex v''}
        pre $lookup((v', v'')) \wedge \nexists v' : lookup((v, v'))$
        let $R = \{((v', v''), w) | \exists w : ((v', v''), w) \in E\}$
    effect (R)
        affected particles {vertex v'}
        $E := E \setminus R$
fraction (particles Z) : payload G
    let $G.V = \{(v, w) \in V | \text{vertex } v \in Z\}$
    let $G.E = \{((v', v''), w) \in E | \text{vertex } v' \in Z\}$

More generally, the set of particles can be defined using criteria on the value, but not the state, of the particles. For example on a set CPRDT holding integers, we can define a criterion as "all the elements greater than 2" but not as "the first 10 elements greater than 2", because it requires knowing which elements greater than 2 are in the set (i.e. their state), and being dependent on the state also means that this set can change with updates.

5.6.2 Shard query

Shard queries are operations that take a specific CRDT version and give the set of particles (the shard) needed to satisfy that query, that can be used to make a partial replica. We already described the fraction operation which creates a partial replica that contains a specific set of particles, but shard queries can be more general than that.

There are two types of queries: state-independent, that only depend on properties of the particles, not the associated state, and state-dependent queries, for which the result depend on the version of the CRDT the query is applied to. For example on a set of integers, a state-independent query could be "the particles greater than 0", while a state-dependent query could be "the particles of the 10 highest integers in the set". The first one does not depend on the state of the CRDT, the result of the query will always be the set $[0, +\infty]$, the second one however will have a different result depending on which elements have been added (and removed if the CRDT supports this operation) on the version considered.

State-independent queries are easier to work with: you can compare them with each other to see if one is more specific than another without having to know the state of the object they apply to. While with state-dependent queries, you can only compare queries if they apply on the same version of the object, otherwise the result may not be exact.

5.6.3 Example

We can put these CPRDTs in pratice on a simple example: the user wall of a social network. Let's assume each replica is associated to a user. So for example we have a OR-set CPRDT for the wall of Alice, and Bob (a friend of Alice), Charlie (a friend of Bob but not of Alice) and an anonymous user who want to look at the posts on the wall or post a message. We assume the replica associated with Alice is full: Alice cares about all the posts on her wall.

Each post contains a date, an author, a message and and a privacy setting, (public, friends of friends, friends only) with regards to Alice instead of the author for simplicity.

The first application of CPRDTs on this example allows to limit the size of the wall: Alice has been using the social network for a few years and there are many posts on her wall. Even if Alice has them all, other users should not have to replicate the whole wall to just look at the latest posts.

Without CPRDTs, one might split the CRDT manually according to some criteria (e.g.: by date, by author, by privacy setting) and then only replicate the CRDTs they care about. While possible, it makes the application more cumbersome to write as it must handle the splitting, new CRDTs must be created according to the criteria, it also makes it harder to work on the non-split CRDT, as it either is another copy or it must be recomposed from the other parts. Even with just one criterion, like the date, it requires to choose an arbitrary splitting scheme: by day, by week, by month ?

With CPRDTs, the splitting is abstracted from the application, each replica can define the posts (particles) it cares about according to its criteria, independently from the other replicas. So Bob might want to look at the posts from Alice and himself made during the last week, while Charlie wants to see all the posts on her wall from two years ago.

Another use of CPRDTs in this example is more security oriented: the posts have a privacy setting, and we would like users to only replicate posts which they are allowed to see. An anonymous user could only replicate public posts while Charlie could also replicate "friends of friends" posts. Of course there needs to be an authority to enforce these limits, it could be a central authority, or in this case Alice could act as the authority, which would for example forward all the updates (using operation based replication) to only the allowed parties.

5.6.4 Shard management

The system model so far has some drawbacks and complexities: the shard of a replica is assumed not to change, operations need to carry additional metadata to be applicable at all involved replicas, and we assume all updates are delivered to all replicas. We have to take into account that the application will need different parts of an object over time, so the shard of the replica should change.

Centralised system model

The model can be greatly simplified if we assume there is a logically centralised entity (which we will call server) holding a full replica and distributing the updates or sharing the new states of the other replicas (within clients), as in SwiftCloud.

First the clients can discard their (partial) replicas at will as long as their updates have been reliably sent to the server. Secondly a client can request any fraction of the full replica to the server to get a new partial replica, or to grow its own shard. The next section describes how to change a shard without losing convergence. Also the updates can be applied at the server's replica, without any need of metadata, and forwarded only to the clients which want them, with added metadata if needed.

The server could also allow making searches on its full replica and transforming them into a shard and its associated partial replica to send to the client. This allows for state-dependent shard queries.

Dynamic change of a shard

Care must be taken if we want to allow the shard to change. Indeed, for operation-based replication, *updates* that do not affect the shard are dropped, which means we cannot simply grow the shard, tell the other replicas about the change, and expect the state to converge, as there will be missing updates. If each replica keeps a log of updates for its own shard, and if we can tell which updates have been applied at a replica and which have not, then there is a possibility to grow the shard. If a replica wants to grow with the set of particles, it can ask other replicas for their log of updates that affect these particles. It can then apply all the updates in the log that he has not yet applied. It can then continue as before but now accepting updates on its new shard. There can still be an issue if an update affects two particles and a replica has one in its shard and the other in the set of the particles it wants to grow, as this update is already applied but not on the extended shard. To avoid this, we assume that an *update* can be split into parts of the update that each only affects one particle. With these parts of update, the replica can apply only the part that affects its shard, and then apply the rest of the update when it grows the shard.

Shrinking the shard of a replica is simple with operation-based replication, it can be done locally when all updates have been handed to the broadcasting subsystem. We also have to make sure that some other replicas have the particles the replica dropped from its shard, as otherwise some information might disappear.

With state-based replication, growing the shard is easier. A replica can tell others to send a larger part of their state, including the set of particles it wants to add, and then merge the incoming state with its own to grow the shard. Note however that the state of the added particles might not be directly causally consistent with the existing state, because the replica might receive a state which is causally before the one it has. If we want to avoid this, the replica has to wait for a state concurrent with, or causally after its own state before growing its shard.

To allow shrinking the shard with state-based replication, a replica must first broadcast its payload *reliably* to ensure all updates indirectly present in the payload will be received by all the replicas.

5.7 Implementation within SwiftCloud

In this section, we describe how we implemented partial object replication in SwiftCloud which was presented in chapter 3. We thus move away from the system model given at the beginning of this chapter. SwiftCloud uses `operation-based replication`, so we do not consider state-based replication in this section.

Currently, replication is available at the CRDT level: the scout can replicate CRDTs as they are used by the application. However these CRDTs can be quite complex and large, especially with collection data types such as sets and graphs, and the CRDT must be fully

replicated even if only a part of it is needed by the application. This can have a high bandwidth and memory cost with large CRDTs. We thus extend this partial replication to a finer grain within the CRDT, using CPRDTs.

Aside from the diminished cost in size, partial CRDT replication has other advantages: it allows to hand off some of the computation of search queries, within the object, to the DC, and it may be used to enforce security restrictions on the data within an object. It also make blind updates available for data types that support them.

5.7.1 External API

The API allows to create transactions and commit or roll them back. These methods are left unchanged, we are only interested in what happens within a transaction. There is a `read(txn_handle, id)` method which returns a view of the CRDT *id*, that is a (full) replica of the CRDT at a version causally consistent with the previous transactions and get calls. *Query* and *update* operations can then be applied on the view until the end of the transaction.

To allow for partial CRDT replication, the SwiftCloud's client API is extended, as shown in Figure 5.1. A new parameter is added to the `read(txn_handle, id)` call to request the CRDT identified by *id* within a transaction: the *lazy?* boolean parameter which means the object's view will not be fully replicated right away from the cache or the DC. When the parameter is *false*, it works as before: a full object replica is given to the application. When it is *true*, a hollow view is created locally and the parts needed for the transaction are lazily loaded when *update* and *query* operations are applied to the view. So, the shard of the local view starts empty and grows with each *update* or *query* operation by adding the *required particles* of the operation to the shard.

This method has the main advantage of being transparent for the application developer: he only needs to change the *lazy?* argument, the rest of the methods stay the same, although behind the scene the replication mechanism is different.

```
begin () : tx_handle
read (tx_handle, object_id, lazy?) : object
fetch (tx_handle, object_id, shard_query) : object
query (tx_handle, object, query_op) : void
update (tx_handle, object, effect_op) : void
commit (tx_handle) : void
rollback (tx_handle) : void
```

Figure 5.1: Extended client API

In some cases we may need to apply many operations on different parts of the CRDT within the same transaction, and with the lazy option this can result in many remote

requests to the DC. To avoid this issue, a special `fetch` operation is added which allows the application to explicitly ask for the parts it will need for the following operations. This operation removes some of the transparency and should thus be avoided, which may be done by combining the operations into a more complex one in the CRDT specification.

5.7.2 Shard query

As explained in section 5.6.2, there are two main types of queries, the state-independent and state-dependent queries. Thanks to the datacentres which hold full replicas of all objects, state-dependent can be applied at the DC when not enough information is available in the cache.

The `shard_query` parameter of the `fetch` API call is defined as follows. It is a function taking a specific version of a full CRDT and returning a partial replica along with its shard definition.

If the shard query is *state-independent*, it has additional properties. First, applying the shard query to any version of a CRDT will give the same shard definition. So for any *update operation* u on a CRDT version v, we have that $shard\_query(v) \bullet u \equiv shard\_query(v \bullet u)$, where the $\bullet$ operator applies the update to the replica. This also implies that a *state-independent shard query* can always be compared to a *shard definition* to know whether the shard query is available in the shard. For example, on a set of integer, a state-independent shard query can be to take all the odd positive integers. If a shard contains all positive integers, then we know the query is available. If it only contains integers greater than two, we know the shard query is not available. This allows to locally verify if a state-independent shard query is available in the cache, or if we need to contact the store.

State-dependent shard queries are not comparable with shard definitions, but they can sometimes be compared with other shard queries. On the integer set example, a state-dependent query can be to take the highest 10 integers. This is easy to compare with other similar shard queries such as taking the 5 highest integers, or the 15 highest integers. We allow the implementer to define whether or not a shard query is *equal or more specific* than another. This makes caching possible: if a user makes one query, then later makes a more specific one, we do not need to fetch a replica again. Note that however the comparison is only relevant if the queries are applied to the *same versions* of a replica.

There are system shard queries, available for all CRDTs, which are the hollow query (no particle), the full query (all particles), and the request of an explicit set of particles.

5.7.3 CRDT specifics

The `fetch` method is available within the CRDT and is how the update and query operations can ensure that the current view has the parts needed for the operation to be

applied. It is the CRDT designer's responsibility to fetch the parts before applying an operation. This corresponds to the required particles given in the specifications.

If lazy capabilities are not supported for a CRDT, then there is only one particle which is the entire CRDT. So a replica can either be hollow or full. This might still be useful because it allows to easily do *blind updates* (updates without knowing the state of the CRDT).

5.7.4 Internal changes

Transaction/client level

The transactions now have to support the added lazy parameter.

When the client does a lazy read in a transaction, a hollow view of the CRDT is fetched. The fetch is local (does not involve the DC).

The transaction handler implements the fetch operation, which checks whether the local version already has the parts for the requested shard query by comparing the query with the shard definition, and if not (or if it cannot do the check locally), it forwards the request to the scout. The scout returns a CRDT view which contains the requested parts, and this view is merged with the local one. The merge is a state merge, which is different for each CRDT, but with relaxed properties: it is only a merge between two views with the same version. Except that because the local view might have non committed updates then on the overlapping particles the local particles must be chosen over the remote ones.

One must also be careful when dealing with blind updates, that is updates that can affect particles without needing them (thus without fetching them). A blind update might be registered by the transaction on a view which does not have the affected particles, and a subsequent fetch could add these particles to the view. Since the fetched particles do not include the update, this update should be applied on those particles. We support this by keeping a log of updates that were not fully applied (some particles affected by the update were not in the view) and by applying these updates to relevant fetched CRDT shards before merging. Figure 5.2 shows an example on an OR-set. The application requests a set, and it receives a hollow view. It then does a blind $add(1)$, which is kept in the log of updates of the transaction. The following $lookup(1)$ require the particle 1, which is fetched from the cache and added to the local view. Since the blind update has only been saved, it is applied now, so the lookup returns true as expected.

Scout/cache level

The scout checks if the requested shard query is available in the cache in a similar way to the transaction level (by comparing it to the shard definition in the cache). This method of caching only works for state-independent queries, which is why there is a second way to check the query's availability: the cache holds the history of the shard queries which

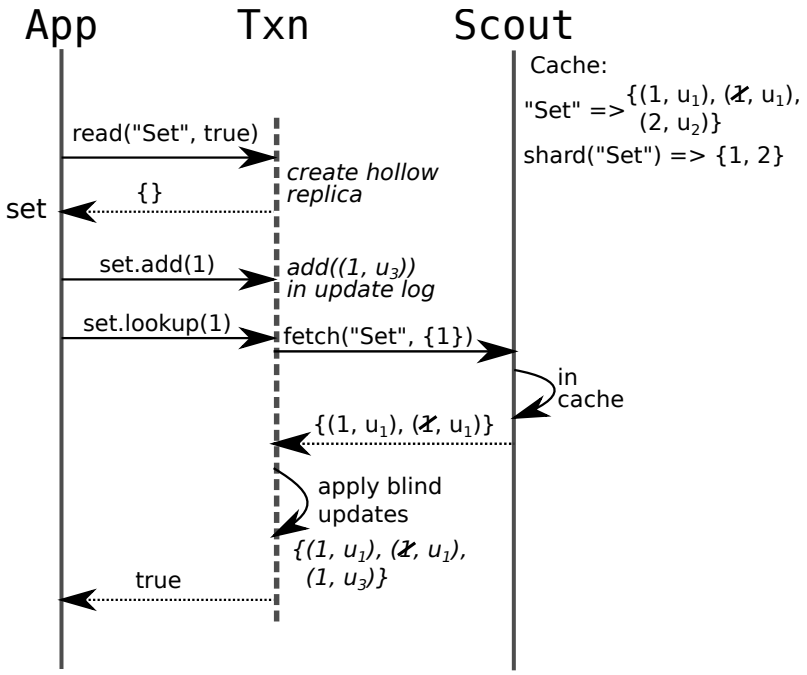


Figure 5.2: Example of a lazy fetching sequence on an OR-set, with the object already present in the cache. *App* is the application and *Txn* is the transaction level. The transaction is assumed to have already been created.

were previously made (and on which version they were made), those shard queries can also be compared to the requested shard queries. But, even with this state-dependent query cache, many requests result in a cache miss because the requested version is slightly different than one on which the shard query was applied. To lower the number of cache misses, an optional time based caching mechanism is implemented: a timestamp is kept for each shard query in the history, and a new request will hit the cache if the same (or a more general) shard query was made soon before, with a configurable threshold.

If the shard query is not available in the cache, the scout forwards the request to the DC. When it receives a partial CRDT in response, it returns the correct view of it to the transaction level. It also merges the partial CRDT with the one currently in the cache.

The merge is more complicated at the scout level: the two CRDTs to merge contain multiple versions, which are represented as a prune point (the state of the CRDT at a pruned version) and an update history (all updates after the pruned version). The DC might have updates that need to be applied to the cache, even if they do not involve parts that were requested. Thus, when requesting a newer version of an object than the one present in the cache, the scout also requests newer updates that affect the shard of the object in the cache. In this way, all the parts of the object in the cache can be updated.

Data Centre level

At the DC, the requested object is fetched, in its full form, and then the shard query is applied on it.

For state-independent queries, the query is simply applied to the pruning point. But for state-dependent queries, the query is executed at a view of the requested version, to find which parts were actually requested, as the query can be complex and depend on the version, but it is applied on the pruning point, to reduce the size of the state. The update history is also pruned from all updates which do not affect the resulting shard and that are not needed to update the cache.

Chapter 6

Evaluation

We evaluate SwiftLinks with a benchmark that simulates typical operations of users: looking at the posts on the frontpage and other pages, reading the comments of a post, posting a link or a comment, and placing a vote on a link or a comment. For each benchmark, the application is prefilled with data that is distributed as real data from the reddit website. By default, unless otherwise noted, it is filled with 10000 posts over 20 forums (so an average of 500 posts per forum), this corresponds to less than a day's worth of data from the actual website. Each post has 20 comments on average. Posts have an average of 170 votes while comments only get an average of 13 votes. These values have been computed from real data from reddit, retrieve through its API, on which SwiftLinks if based.

During the benchmark, just 10% of the operations are *writes* (posts, comments, or votes), the rest are *reads*. Also, 90% of operations are biased and are made on previously read posts and comments, such that they are likely to hit the cache, the rest are done on randomly selected posts and comments.

6.1 Implementation specifics

There are two main types of CRDTs in the implementation: the set of posts and the trees of comments. These two types are combined with vote counters, so each element or node also holds a vote counter to know the score attributed to posts and comments. They are combined because the vote counter is needed to show the scores and also to sort the posts and comments on the client machine.

The read operations use *state-dependent* queries. For reading links, a query requests the first 25 links after a specific link, or at the beginning of the set, according to an ordering described in Appendix A. When reading a comment tree, only the top 20 comments are queried. Both of these queries are state-dependent as both which links or comments are present and the number of votes influence its result.

6.2 Experimental setup

SwiftLinks was evaluated using three Amazon EC2 servers as datacentres. One in Ireland and two in the USA, near the east coast and the 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 which is shared between multiple users. There are five reddit users running concurrently per node. Each user does one operation each second.

There are three main configurations for clients to run the application, *Cloud*, *Lazy*, and *Non-lazy*.

Operations can be applied synchronously at one DC and replicated asynchronously, which we call *Cloud* mode. This is done by running a server within the DC, and executing operations at clients through remote procedures calls. This simulates typical geo-replication at multiple DCs. All the application logic runs at the server, the client only call the server and returns the result of the operation, it does not retrieve the internal state of the CRDT. In this case, the client has no cache at all.

The other two configurations adopt the SwiftCloud approach of co-locating a scout at each client thread with varying maximum cache sizes, 64 MegaBytes by default. If the cache size exceeds this limit, the least recently used object is dropped from the cache and must be fetched again from the remote store. This simulates memory limits on mobile devices. The application logic is run locally at the client on the cached objects and committed asynchronously to the store.

The *Lazy* configuration benefits from the partial replication mechanism described in the previous chapter, which means objects are fetched in parts as needed, so the cache can hold partial objects. In the API, the `lazy?` argument is set to *true* with the `read` API call.

The pre-existing SwiftCloud caching method is used in *Non-Lazy* mode, in which the cache only holds full objects, with a shard π. These objects are fetched completely when needed. The `lazy?` argument of the API is thus set to *false*.

6.3 Latency

We evaluated the perceived latency for various operations with and without partial object replication. Figure 6.1 shows the cumulative distribution functions of different operations' latency with a 64MB cache size limit. These results are with a warm cache, that is the cache has already been filled with objects needed by the application which were fetched by previous operations.

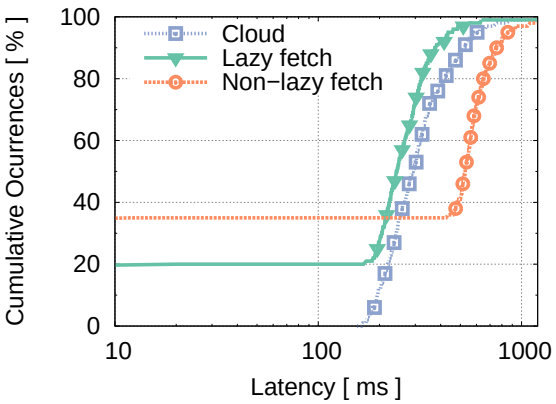
In Figure 6.1a, we see that the non-lazy mode has a more than 35% cache hit rate as this fraction of operations has almost no latency. The hit rate is not optimal due to

the limit in cache size: the cache cannot hold full replicas of all the forums and thus sometimes need to fetch them again. Figure 6.2 shows the same results but without any cache size limit. In that case, the cache hit rate is of 90%, which corresponds to our ratio of biased operations, and it confirms the previous results with a social network application of the SwiftCloud paper [26]. With lazy fetching, the cache hit rate is lower, with only 20%, because the cache only holds partial replicas which gives it less chance of having all the parts needed for reading a page. However, it has the advantage of a lower maximum latency: if an operation does not hit the cache, it only needs to fetch the parts it needs from the store, instead of the full object. So in that case it induces a delay similar to the cloud solution, around 200 to 300 milliseconds, while without lazy fetching, the delay is increased to around 500 to 700 ms by having to replicate a full object. This gives a trade-off between the cache hit rate and the maximum latency. While fully replicating an object will provide more cache hits, if the object is large a cache miss is more costly and only partially replicating reduces latency.

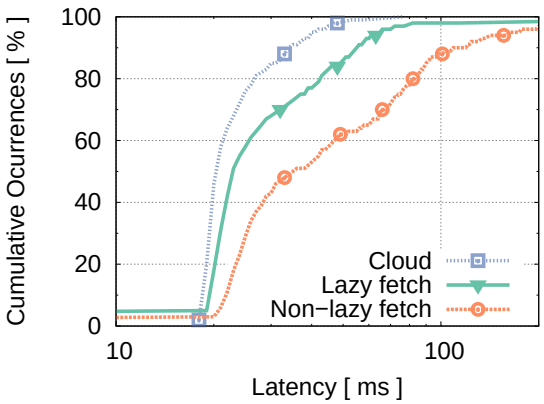
For the latency of reading comments of a post, shown in Figure 6.1b the situation is a bit different. As the cache is separate for each client, the operation is less likely to hit the cache as a user rarely reads the same comment tree multiple times. This translates into a small cache hit rate, of less than 5% with both lazy and non-lazy fetching. But again, lazy fetching has the advantage of reducing the impact of a cache miss as it only replicates the comments required by the operation instead of the full comment tree of a post. So that approach has a slightly better latency, close to the cloud mode. Cloud mode has an advantage over Lazy fetching because it does not need to retrieve the internal state of the CRDTs, which means the returned messages are smaller. The difference with non-lazy fetching is not as large as when reading pages of links because the objects involved are smaller.

Update operations are where caching is the most beneficial in this application. Indeed, update operations are typically applied on objects, or parts of objects, that have already been read by the client. Also, the update operations use state-independent queries to fetch their missing part, which is easier to compare with what is in the cache. Figure 6.1c shows this is indeed the case. While the cloud mode has an almost constant latency for all operations of a round-trip time, with cached modes most of the operations, almost 90%, have no latency. Again, the lazy mode has the advantage of reducing the latency when the cache is not hit, as it only needs to fetch the part of the object that needs to be updated instead of the full object. Moreover, some updates can be done blindly, which means it can be done totally locally.

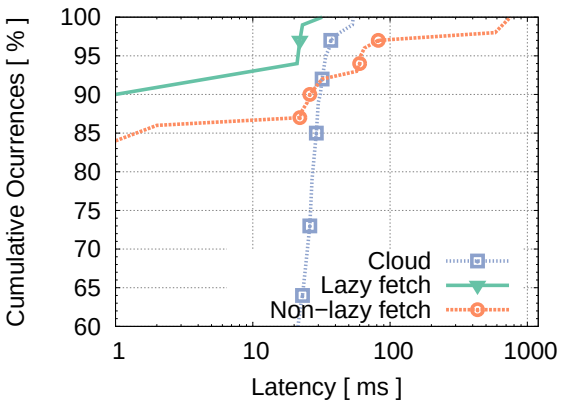
In particular, Figure 6.3 shows the benefit of updates when posting comments, which almost only require particles already present in the cache. You can see that with lazy fetching, all the operations have almost no latency, as they can be done completely asynchronously, while with non-lazy fetching there can be a large delay when the tree of



(a) Reads of pages of links



(b) Reads of comments of a link



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

Figure 6.1: Perceived latency of SwiftLinks at one site with medium (64MB) cache size limit and a warmed up cache.

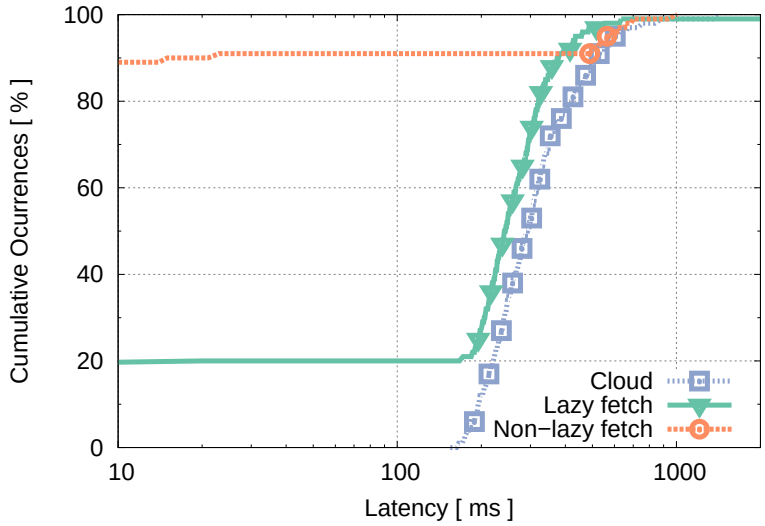


Figure 6.2: Reads of pages of links with unlimited cache which is already warmed up.

comments is not in the cache, as it needs to be fetched from the store. Even if an operation cannot be done completely locally with lazy-fetching, then the client only has to fetch one node of the tree to be able to apply the update, instead of the complete tree.

6.4 Impact of cache size limit

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

6.4.1 Impact on latency

We have already seen that non-lazy fetching performs much better without cache limit when reading links. Figures 6.4 and 6.5 show the same plots as Figure 6.1, but with cache size limits of 16MB and 128MB, respectively.

As shown in Figures 6.4a and 6.4c, a smaller cache size limit has a big latency impact on reading links and updates with Non-lazy fetching, while its impact is much lower with Lazy fetch. With a small cache, the cache hit rate of Non-lazy fetch of reading links becomes worse than Lazy fetch, as only a few objects can fit in the cache at a given time and thus need to be fetched much more frequently. This results in a lower fraction of operations having no latency than with a 64MB cache, about 5% against the previous 35%. There is also an impact for Lazy fetch, but it is much lower: it only drops to 13% from 20%. The same is true for update operations, Figure 6.4c shows that Lazy fetch performs significantly better than Non-lazy with a small cache.

Reads of comments are almost not impacted by the cache size limit: the operations have

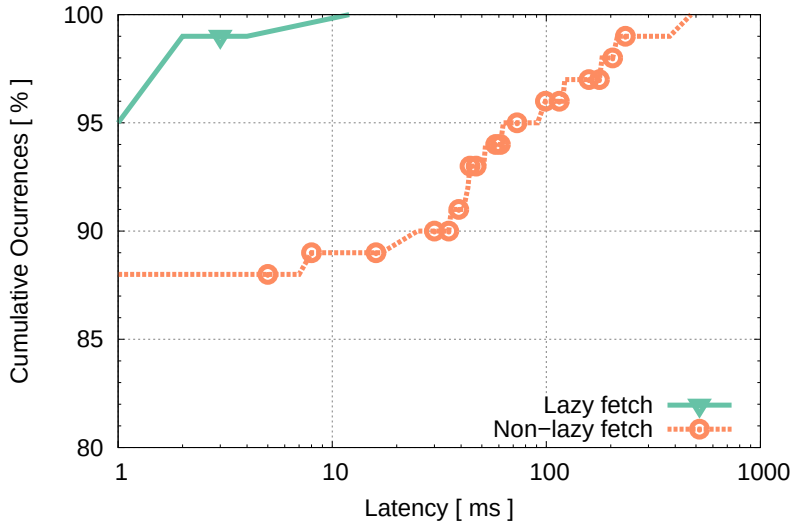


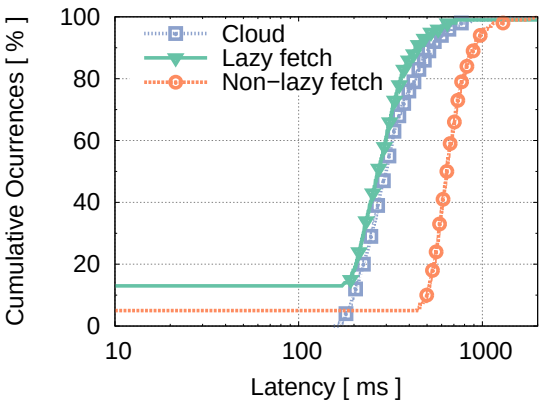
Figure 6.3: Perceived latency of commenting on a post, at all sites, with medium (64MB) cache size limit.

a low cache locality, so most operations need to fetch a replica from the DC. Figures 6.4b and 6.5b show the perceived latency with a 64MB and 128MB cache size limit, respectively. The results are quite similar, with Lazy fetch better than Non-lazy, and Cloud better than Lazy as it does not fetch metadata, except for around 3% of the operations which hit the cache.

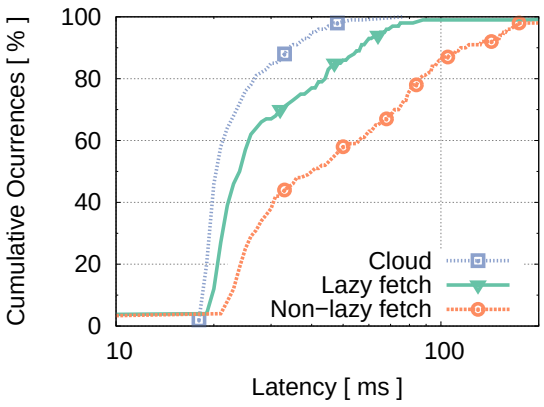
With a 128MB cache size limit, as shown in Figure 6.5a, Non-lazy fetch has a large portion of zero latency operations when reading links, as more link sets can be kept in the cache. It however still performs worse than Lazy fetch for operations that do not hit the cache. The latency of update operations is also improved for Non-lazy fetch with a bigger cache (Figure 6.5c), but Lazy fetch still outperforms it for the same reasons as before.

6.4.2 Impact on cache miss rate

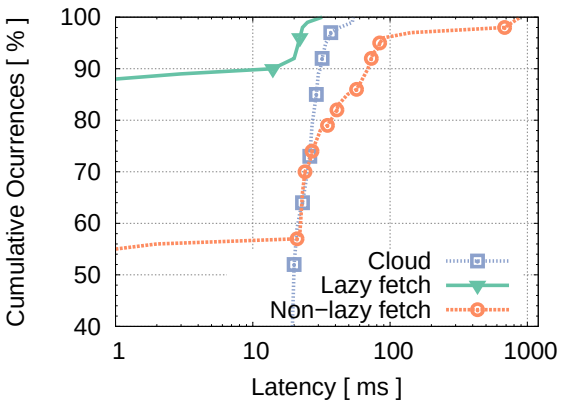
The size limit imposed on the cache has an impact on the cache hit rate. While lazy fetching means that there are more cache misses, they are less costly as shown in the previous section. We show how lazy and non-lazy fetching are impacted in Figure 6.6. The performance of the cache with lazy fetching is less impacted by the size limit, the 16MB and 128MB cache experiments show almost the same number of around 180 cache misses. On contrary, with non-lazy fetching, as the cache is quickly filled and some objects must be dropped, the number of cache misses decrease with the size of the cache.



(a) Reads of pages of links

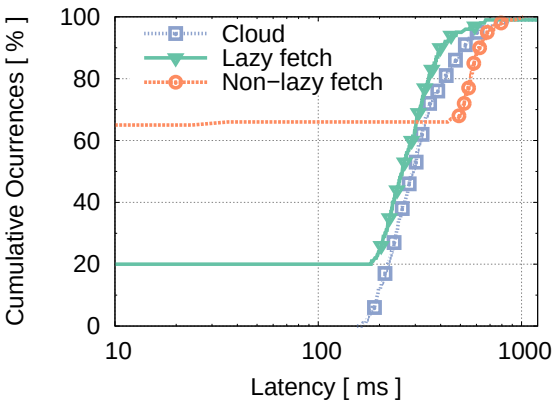


(b) Reads of comments of a link

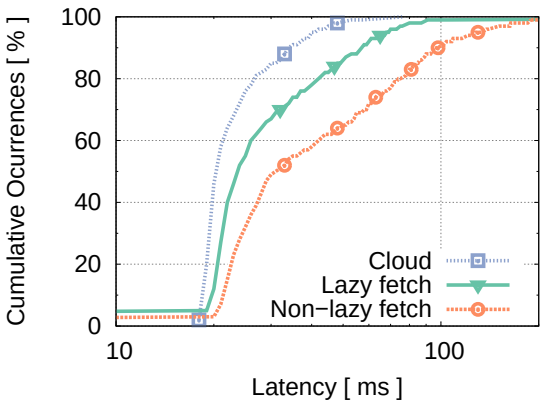


(c) Updates: posting a link, commenting, voting a link, and voting a comment

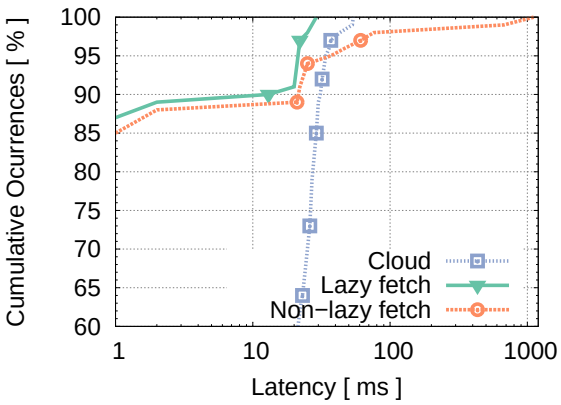
Figure 6.4: Perceived latency of SwiftLinks at one site with small (16MB) cache size limit and a warmed up cache.



(a) Reads of pages of links



(b) Reads of comments of a link



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

Figure 6.5: Perceived latency of SwiftLinks at one site with large (128MB) cache size limit and a warmed up cache.

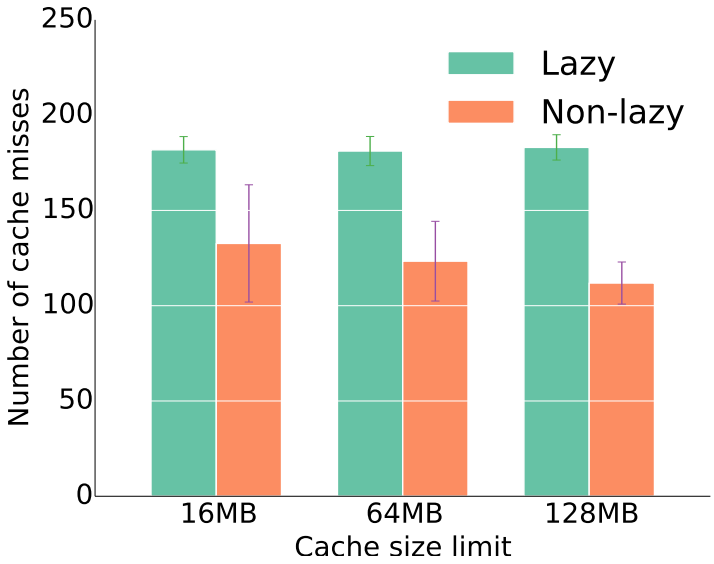


Figure 6.6: Number of cache misses with different cache size limits.

6.4.3 Impact on number of objects in the cache

Another impact of the cache size limit is the number of objects that can be kept in the cache. Figure 6.7 shows the difference between lazy fetching and non-lazy fetching. For partial replication, only one object is counted even if multiple parts of it have been fetched over time. In the lazy mode, many more objects can be in the cache at any moment since only parts of the object are kept. 64MB is enough to keep all the objects needed by the application, while in non-lazy mode, even 128MB is not enough, with around 40 objects versus 100 with partial replication.

6.5 Bandwidth usage

As shown in figure 6.8, partial replication can reduce significantly the bandwidth usage of fetching objects. It measures the average bandwidth usage of one client over one minute, with the cache already warmed up.

6.6 Cache warm up

The results shown previously are taken with the cache warm. In practice if we want the cache to be on the client device, and not on servers from the application infrastructure, we must consider the performance when the cache is still cold, which happens when starting the application.

Figure 6.9 shows the latency of operations during the first 10 seconds of running the application, with a cold cache. In this case, lazy fetching gives lower latency as it does not need to replicate the full object. The difference is especially strong for links

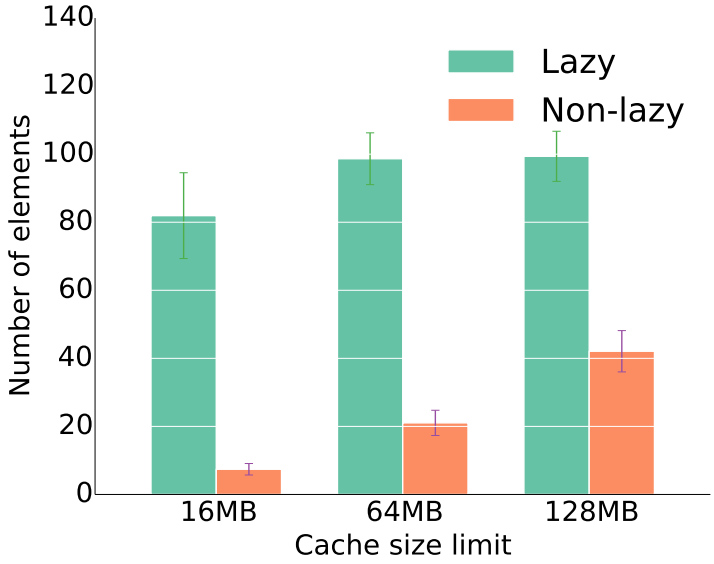


Figure 6.7: Number of objects kept in the cache during a benchmark with or without lazy fetch. In lazy mode objects can be *partial*, which in non-lazy mode all objects are *full* replicas.

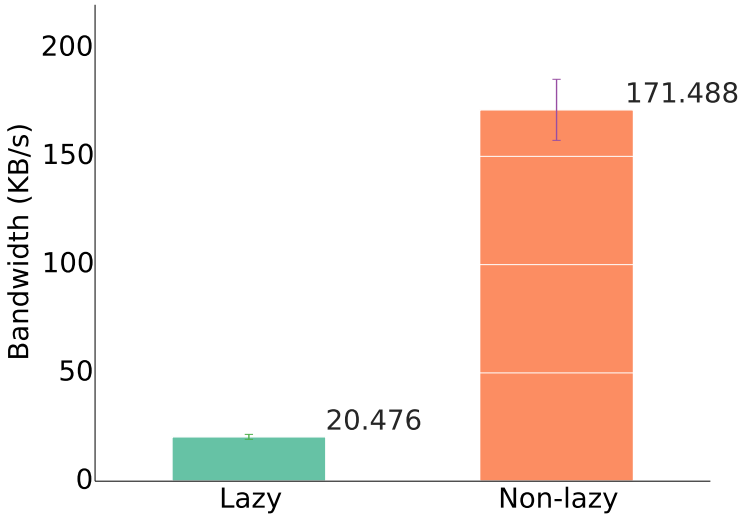
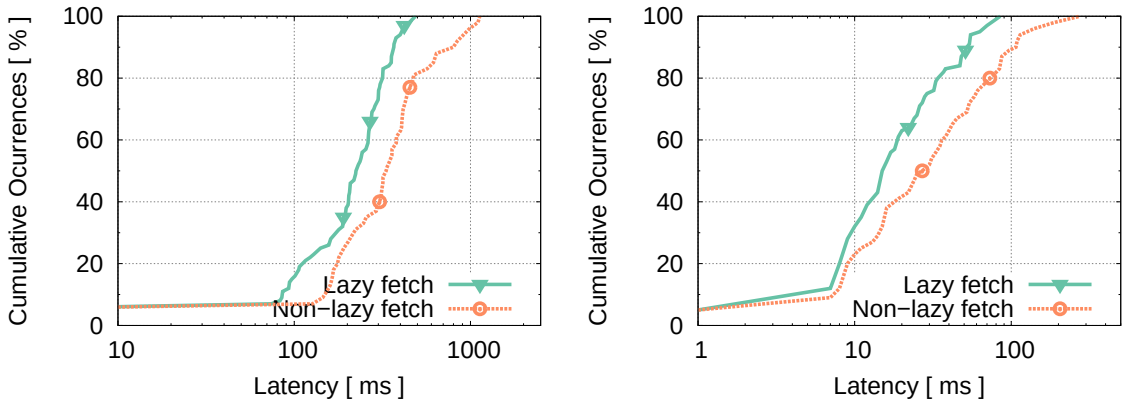


Figure 6.8: Average bandwidth usage to fetch objects with a 128MB cache limit, with the cache already warmed up.



(a) Reads of pages of links

(b) Reads of comments of a link

Figure 6.9: Perceived latency of SwiftLinks at one site during cache warm up.

reading operations, as shown in Figure 6.9a, as the set of links are large objects. But even for smaller objects, such as comment trees, lazy fetching improves the performance (Figure 6.9b). Also note that the cache size limit has no impact for cold cache performance: the cache is not yet filled after 10 seconds, so no object is evicted from it during that period.

6.7 Discussion

We have seen that lazy fetching has advantages over full replication of objects. It puts a upper bound on the latency of operations by limiting the size that is fetched from the store, which gives performance at worst similar to a cloud based server architecture while keeping some of the benefits of client-side caching.

Blind update operations gain the additional benefit of being applied locally even if the object is not in the cache.

It also limits the memory usage of the cache, which allows more objects to be kept locally even with a small cache size limit. This is useful for memory-thin devices, and to work on very large data structures with a low memory usage.

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

The last advantage is a lower cost of filling the cache when starting the application. When the cache is empty all operations induce a cache miss, which is especially costly if a large object has to be fetched. Lazy fetching limits this issue by only replicating the parts of the object that are actually needed.

Lazy fetching has one main drawback, it limits the cache hit rate as an object is not fully replicated right away, and parts may need to be fetched from the store over time. So it should be used when the cost of a cache miss with full replication outweighs the cost

of reduced number of cache hits. A trade-off is possible between the two: instead of only fetching the parts needed by the application, we could fetch more parts of the object in order to improve the cache hit rate. This would however increase bandwidth and cache size utilisation. Latency could be kept low by doing this additional fetch asynchronously, when the user is not doing any operation.

6.8 Use cases of lazy fetching

The results of the evaluation highlights two cases where lazy fetching is especially suited.

The first is when working with large CRDTs. In this case, lazy fetching significantly reduces the latency induced by cache misses, it reduces the memory usage of the cache, and it makes updates always fast.

The second is when the operations have a low cache locality: fetching full objects is counter productive as there are a lot of cache misses and the cached objects are rarely used. On the other hand lazy fetching can give performance close to cloud-based solutions while keeping the advantages of caching for update operations.

Chapter 7

Conclusion

We presented the concept of conflict-free partially replicated data types, an extension of CRDTs which allows replicas to hold parts of larger data structures. It optimises the bandwidth and memory usage of replicas by only replicating elements needed by the application, and makes blind updates possible. We explained how state-based and operation-based replication mechanisms must be adapted to support partial replicas. We showed how to specify those CPRDTs by building upon previous work and we gave examples of data structure specifications. We extended SwiftCloud to support partial object replication as an additional form of client-side caching.

We built an application based on reddit, called SwiftLinks, and designed the novel CPRDTs it required.

With this application, we evaluated partial replication against both SwiftCloud's existing client-side caching and traditional datacentre-only geo-replication. We modelled our benchmark with data retrieved from the actual reddit website. Our evaluation showed that partial replication gives latency and bandwidth usage benefits over previous schemes, especially when objects used by the application cannot be all fully kept in the cache, or when starting the application with an empty cache.

7.1 Future work

Future work will evaluate the lazy fetching solution on different applications to get a better view of its benefits and drawbacks. Partial replication can be used as a security mechanism to avoid replicating sensitive data by restricting access with finely grained rules. Improving the caching heuristics with predictive caching to fetch object parts ahead of time would reduce the cache miss rate.

Bibliography

[1] Alexa Internet Inc. reddit.com site overview. `http://www.alexa.com/siteinfo/reddit.com`. Accessed: 2014-06-02.

[2] Bailis, P., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 761–772.

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

[4] Fivecoate, M. Surfacing interesting content. `http://stdout.heyzap.com/2013/04/08/surfacing-interesting-content/`. Accessed: 2014-06-15.

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

[6] Guerraoui, R., and Rodrigues, L. *Reliable Distributed Programming*, vol. 138. Springer, 2006.

[7] Gustavsson, S., and Andler, S. F. Self-stabilization and eventual consistency in replicated real-time databases. In *Proceedings of the First Workshop on Self-healing Systems* (New York, NY, USA, 2002), WOSS '02, ACM, pp. 105–107.

[8] Jay, C., Glencross, M., and Hubbold, R. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact. 14*, 2 (Aug. 2007).

[9] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 265–278.

[10] Linden, G. Marissa mayer at web 2.0. `http://glinden.blogspot.be/2006/11/marissa-mayer-at-web-20.html`. Accessed: 2014-06-23.

[11] Linden, G. Make data useful. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf, Nov. 2006.

[12] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[13] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Stronger semantics for low-latency geo-replicated storage. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 313–328.

[14] Munroe, R. reddit's new comment sorting system. http://www.redditblog.com/2009/10/reddits-new-comment-sorting-system.html. Accessed: 2014-06-15.

[15] Navalho, D., Duarte, S., Preguiça, N., and Shapiro, M. Incremental stream processing using computational conflict-free replicated data types. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms* (New York, NY, USA, 2013), CloudDP '13, ACM, pp. 31–36.

[16] reddit inc. About reddit. http://www.reddit.com/about/. Accessed: 2014-06-02.

[17] reddit inc. reddit source code. https://github.com/reddit/reddit. Accessed: 2014-04-08.

[18] Salihefendic, A. How reddit ranking algorithms work. http://amix.dk/blog/post/19588. Accessed: 2014-06-15.

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

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

[21] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 385–400.

[22] Srinivas, D. Reddit ranking algorithm implementation with sql. http://blog.sodhanalibrary.com/2014/04/reddit-ranking-algorithm-implementation.html. Accessed: 2014-06-15.

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

[24] Vogels, W. Eventually consistent. *Commun. ACM 52*, 1 (Jan. 2009), 40–44.

[25] Wittie, M. P., Pejovic, V., Deek, L., Almeroth, K. C., and Zhao, B. Y. Exploiting locality of interest in online social networks. In *Proceedings of the 6th International COnference* (New York, NY, USA, 2010), Co-NEXT '10, ACM, pp. 25:1–25:12.

[26] Zawirski, M., Bieniusa, A., Balegas, V., Duarte, S., Baquero, C., Shapiro, M., and Preguiça, N. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. Rapport de recherche RR-8347, INRIA, Oct. 2013.

Appendix A

Ranking of posts and comments

reddit supports multiple orderings to surface interesting content depending on its votes [22, 18, 4, 14].

A.1 Hot

The main ranking algorithm gives a hotness value to each post which depends on the date of the post and its score.

The function that gives the rank of a post or comment is:

$$f(score, date) = \log_1 0(score) + sign \times \frac{date}{45000}$$

With *date* being the number of seconds it was posted after a specific date. *sign* is simply the sign of the score: $+1$ if score is > 0, -1 if it is < 0, and 0 if it is $= 0$.

A.2 Confidence sort

For comments, a different algorithm is used by default. It instead uses statistical sampling to find the best comments using the Wilson score interval.

$$\frac{1}{1+\frac{1}{n}z^2}\left[\hat{p}+\frac{1}{2n}z^2 \pm z\sqrt{\frac{1}{n}\hat{p}(1-\hat{p})+\frac{1}{4n^2}z^2}\right]$$

With $\hat{p}$ being the observed fraction of positive ratings, i.e. the number of upvotes over the number of all votes.

With n the total number of votes.

And z is a confidence interval parameter that corresponds to a quartile in the standard normal distribution. For example, 1.0 corresponds to a 85% confidence interval, and 1.6 corresponds to 95%.

A.3 Other

There are other simple algorithm such as newest, oldest, and also top which ranks according to the score.

Appendix B

Particle definition

We define a particle according to the query and update operations available on a CRDT. We assume we already have a correct CRDT specification. We show how to define what is a particle on two data structure interfaces. The dual counter example in Specification 16 is two counters combined: you can query the value of the first or second counter independently, you can increment the first one, but you can only increment the second by incrementing both counters at the same time. The other example, in Specification 17 is a set which supports *add* and *remove* methods, we can also *lookup* whether an element is in the set or not, and ask for the *size* of the set.

Specification 16 Dual counter interface

1: The interface of a CRDT which has two counters.
2: **query** firstValue() : integer
3: Returns the value of the first counter.
4: **query** secondValue() : integer
5: Returns the value of the second counter.
6: **update** incrementFirst()
7: Increments the value of the first counter by one.
8: **update** incrementBoth()
9: Increments the value of both counters by one.

A query operation is the name of the query method, and the value of its arguments. There is thus one query operation for each possible value of its arguments. Similarly, an update operation is the name of the update method, and the value of its arguments.

For a specified CRDT, we can define a set Q, consisting of all its query operations, and a set U which contains all its update operations. On the dual counter example, there are two query operations, `firstValue()` and `secondValue()`, and two update operations, `incrementFirst()` and `incrementBoth()`. On the set example, there is the `size()` query operation, and as many `lookup` operations as there are possible elements in the set. There is also as many `add` and `remove` operations as the possible elements of the set.

Specification 17 Set interface

1: The interface of a typical set CRDT.
2: **query** size() : integer
3: Returns the number of elements in the set.
4: **query** lookup(element e) : boolean
5: Returns true if e is in the set.
6: **update** add(element e)
7: Adds the element e to the set.
8: **update** remove(element e)
9: Removes the element e from the set.

We can map each query operation q to a set of update operations that might modify the result of the query operation with a function affecting(q). More formally, an update operation u is in affecting(q) if there exists a valid execution of updates where, at a state s, result$(s, q) \neq$ result$(s \bullet u, q)$, where result(s, q) gives the result of the query operation q on state s.

On the dual counter, affecting(`firstValue`()) = {`incrementFirst`(), `incrementBoth`()} and affecting(`secondValue`) = {`incrementBoth`()}. On the set, affecting(`size`()) = U, and, for any element e, affecting(`lookup`(e)) = {`add`(e), `remove`(e)}.

With this mapping, we can form a partial order $\preccurlyeq$ on the queries, such that for query operations q and q', $q \preccurlyeq q'$ if and only if affecting(q) $\subseteq$ affecting(q'), we say q is *more specific* than q'. The set of particles π of a CRDT is equal to the smallest set of more specific query operations for which $\bigcup_{q \in Q}$ affecting$(q) = U$, so this set must include all updates through the affecting function. If we apply this to the dual counter example, we have three possibilities for π: {`firstValue`()}, {`secondValue`()}, or {`firstValue`(), `secondValue`()}. While `secondValue`() is the most specific query operation, it does not include all updates in its affecting set. {`firstValue`()} is thus the set of particles, as it includes all updates and it is smaller than having two query operations. For the set example, π is the set of `lookup` operations, or, if we drop the lookup name, the set of possible elements of the set. `size` is not in π because it is less specific than all other query operations.

B.1 Partial update

For the dual counter example, we would expect two particles, one for each counter, but from our definition there is only one. This is because the `incrementBoth` update affects both queries and there is no update operation only affecting the `secondValue` query. To get two particles, we can consider that some updates can be divided into smaller updates, partial updates, that are always applied together. We give an example for the dual counter in Specification 18. Then, we can replace the complete `incrementBoth()` update

operation in the U set by both partial updates, `incrementBoth():incrementFirst()` and `incrementBoth():incrementSecond()`.

Specification 18 Partial updates for incrementBoth in the dual counter example

```
1: update incrementBoth()
2:     partial update incrementFirst()
3:         Increments the value of the first counter.
4:     partial update incrementSecond()
5:         Increments the value of the second counter.
```

B.2 Particle to internal state

We define the possible internal state of a CRDT as all the resulting states of valid executions of update operations. For state-based replication, this corresponds to the semi-lattice formed by the payload of the CRDT, and for operation-based it is the actual executions of the downstream parts of the update operations. With partial replication, the shard limits the internal state to a set of particles. In particular, one particle p limits the possible states as all the resulting states of valid executions of update operations, with any update operation u not affecting the particle ($u \notin \text{affecting}(p)$) replaced by a no-op.

The possible states for a shard is simply the union of the possible state of each particle in the shard. This mapping can be seen as a definition of the fraction function.

B.2.1 Overlapping state

With the definition so far, two particles can map to overlapping possible states. For example, if we allow the dual counter to also increment its second counter by itself, as in Specification 19, both `firstValue` and `secondValue` particles include the `incrementBoth()` update operation in their affecting set. Thus, the possible state created or modified by this operation will be present with both particles. We would like to avoid this to make particles optimal: the possible state associated to a particle should be as small as possible. We can use partial updates to achieve this. Indeed, if the `incrementBoth()` update operation is split into two updates, one for each counter, then the affecting (possibly partial) updates of each particle are non-overlapping.

B.3 Required particles

In CPRDT specifications, we give *required particles* for operations, which are the particles the operation needs to know the state of to correctly execute. We can specify it formally as such, a query operation q has required particles: $\text{required}(q) = \{p \in \pi | \, \text{affecting}(p) \cap \text{affecting}(q) \neq \varnothing\}$. We can do the same for update operations if we define affecting(v)

Specification 19 Interface of dual counter with independent update operation

1: The interface of a CRDT which has two counters which can be incremented independently.
2: **query** firstValue() : integer
3: Returns the value of the first counter.
4: **query** secondValue() : integer
5: Returns the value of the second counter.
6: **update** incrementFirst()
7: Increments the value of the first counter by one.
8: **update** incrementSecond()
9: Increments the value of the second counter by one.
10: **update** incrementBoth()
11: Increments the value of both counters by one.

on an update operation v. An update operation u is in affecting(v) if there exists a valid execution of updates where, at a state s, $s \bullet v \neq s \bullet u \bullet v$. Note that we assume that no two particles map to overlapping possible states, as otherwise we might have to choose between two particles that both provide enough information for the operation.

B.4 Affected particles

The *affected particles* of an update operation u is affected(u) = $\{p \in \pi | u \in \text{affecting}(p)\}$, so the inverse mapping of the affecting function. If there is no overlapping in the possible states of particles, there should be only one affected particle per (partial) update operation.

Appendix C

Implemented code overview

You will find attached to this text the source code of SwiftCloud's core and of the SwiftLinks application.

C.1 SwiftLinks

SwiftLinks source code is located in the `src-app/swift/application/swiftlinks` directory. In that directory `SwiftLinks.java` implements the API of the application to support all its operations. `SwiftLinksBenchmark.java` implements the benchmark used for the evaluation. In the `cprdt` subdirectory you will find the implementations of the set of links and of the comment tree.

JUnit tests of the application can be found in the `src-app-test/swift/application/swiftlinks` directory.

C.2 Core

Changes were also made to the core to support partial replication.

The transaction system was adapted with additional API calls, you can find the interface at `src-core/swift/crdt/core` in the `TxnHandle` class.

In the same directory, the CRDT basic interface was adapted to support shards, you can see this in the `CRDT` and `BaseCRDT` classes. The internal representation of a CRDT was also extended to allow merging updates from different partial CRDTs. The implementation is in the `ManagedCRDT` class.

CPRDT specifics are in the `src-core/swift/cprdt` directory, with the shard definition class and basic shard queries.

Implementation of the client side support for partial replication is in the `src-core/swift/client` directory. Classes `AbstractTxnHandle` and `AbstractTxnHandle` contain the implementation of the transactions. The large `SwiftImpl` contains the core of the client-side implementation. Changes were made to the `get` methods.

On the datacentre side, with its code located in the `src-core/swift/dc` directory, changes were made to the `DCSurrogate` class in the `handleFetchVersionRequest` method to support partial replication.

C.3 Evaluation

Scripts to create the plots for the evaluation are located in `scripts/groovy/swift/stats`, at files starting with `swiftlinks`, and in `scripts/python/swiftlinks`.

Scripts to run the evaluation on PlanetLab and Amazon EC2 are located in the `scripts/groovy/swift/deployment` directory, in the `runlinks.groovy` and `SwiftLinks.groovy` files.