Dynamic Adaptation of Geo-Replicated CRDTs

(extended abstract of the MSc dissertation)

Carlos Guilherme Crisóstomo Bartolomeu
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—With the advent of cloud computing, and the need to maintain data replicated in geographically remote data centers, searching for strategies to provide data consistency with minimal synchronization became very relevant. Unfortunately, most data types require operations to be totally ordered to ensure replica consistency.**

Conflict-free Replicated Data Types (CRDTs) are data types whose operations do not conflict with each other and, therefore, can be replicated with minimal coordination among replicas. While it is easy to ensure that all replicas of CRDTs become eventually consistent when the system becomes quiescent, different techniques can be used to propagate the updates as efficiently as possible. Different approaches, such as state transfer and operation forwarding, have been proposed to propagate the updates as efficiently as possible, with different tradeoffs among the amount of network traffic generated and the staleness of local information.

This thesis proposes and evaluates techniques to automatically adapt a CRDT implementation, such that the best approach is used, based on the application needs (captured by a SLA) and the observed system configuration. Our techniques have been integrated in SwiftCloud, a state of the art geo-replicated store based on CRDTs.

I. Introduction

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

Conflict-free Replicated Data Types (CRDTs)[2], [3], [4] are data types whose concurrent operations do not conflict with each other and, therefore, can be replicated with minimal coordination among replicas. CRDTs are implemented in such a way that any two concurrent operations A and B are commutative and, therefore, even if they are executed in different sequential orders at different replicas, the final result is still the same. As a result, there are no conflicts among concurrent operations and replicas can often execute operations promptly, without synchronization with other replicas, i.e., operations may be executed locally first and shipped to other replicas only when it becomes appropriated. Using this approach, even if replicas diverge from each other, convergence is eventually reached due to CRDTs properties. Thus CRDTs, unlike other eventual consistency approaches, may strongly simplify the development of distributed application such as social networks, collaborative documents, or online stores[5], [6], [7].

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

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

The rest of this document is organized as follows. For self-containment, Section II provides an introduction to CRDTs and a description of its current implementations. Section III introduces Bendy along with the design and implementation. Section IV presents the results of the experimental evaluation study. Finally, Section V concludes this document by summarizing its main points and future work.

II. Related Work

Considerable work has been done in order to use CRDTs as an advantage by many systems that require high availability. Examples of that are the systems like Dynamo [11], Riak [12] and SwiftClowd [5].

A. Service Level Agreement (SLA)

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

In most cases, applications prefer to use the stronger consistency, when the network conditions are favorable. However, when the network is unstable (higher latencies due to congestion, partitions, etc), different applications require different consistency guarantees. Actually, the preferred consistency guarantee may be even a function of the actual value of the network latency that is observed (for instance, an application may be willing to wait t seconds to get strong consistency but not more).

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

In the context of our project, we will not use the SLA to determine the consistency level to use. Instead, the max time that a user is willing to wait for an update, SLA time, will be used by our system as a hint to determine which approach an object should use.

B. CRDT implementations

Conflict-free Replicated Data Types (CRDTs) are data structures that allow replicas to be updated concurrently and still ensure that all replicas may eventually converge to the same state[2], [3]. CRDTs are a powerful alternative to simpler approaches to reconcile divergent replicas, such as user-specified functions[13], which make programming hard, or last-write-wins semantics[14], that may cause updates to be lost. Two main types of CRDTs have been proposed, based on different propagation models, namely operation- and state-based CRDTs.

For operation-based CRDTs, a replica propagates its applied operations to other replicas. Concurrent operations are designed to be commutative; thus, replicas can deliver concurrent updates in different orders and still converge to the same state, without the risk of having conflicts. This approach requires exactly-once delivery, a quite expensive requirement and, in some cases, causally ordered delivery (see, for instance, the optimized observed-removed set[15]).

On the other hand, for state-based CRDTs, a replica ships its whole internal state to other replicas. Upon arrival of a state update, replicas merge both the local and the received state. The merge operation of state-based CRDTs is idempotent, commutative, and associative. Therefore, state-based CRDTs have less requirements for the delivery channel compared to operation-based CRDTs: messages can be lost, duplicated or even delivered out-of-order, but replicas will converge to the same state as long as they have seen the latest states from each other. To ensure that states can be merged in this manner is not trivial and, usually, the state must be encoded in a manner that is less space efficient than with operation-based CRDTs. Finally, different strategies may be used to decide when to propagate the state of one replica. For instance, a new state can be sent every time a client request is processed, or a new state can be sent periodically, incorporating the result of multiple requests.

C. SwiftCloud

SwiftCloud is a geo-replicated cloud storage system that stores CRDTs and caches data at clients [5]. It consists of several datacenters that fully replicate the datastore. Clients communicate with the closest datacenter and locally cache recently accessed data. To the best of our knowledge, SwiftCloud is the most complete and up-to-date geo-replicated storage system that incorporates support for CRDTs. Therefore, we have opted to use it as a testbed to get more insights on the advantages and disadvantages of operation based and state based CRDTs in practice.

In SwiftCloud, transactions are first executed and committed in the client side, then propagated to the preferred datacenter, which eventually propagates committed transactions to the rest of datacenters (using an operation-based approach). Thus, SwiftCloud is able to provide low access latencies for both write and read operations by delivering slightly stale data. For fault tolerance, committed transactions are only visible to other clients after they have been seen by K datacenters.

The way transactions are propagated and applied provides causal+ consistency. Causal consistency is tracked and enforced by relying on a vector clock with an entry per datacenter. SwiftCloud implementation is based on a per-datacenter serialization point that sequences local updates and enforces causal dependencies.

III. Bendy

Our study have shown that when updates need to be propagated as soon as possible, the operation-based approach outperforms the state-based approach; this justifies the original SwiftCloud implementation. However, when clients are willing to tolerate reading information that is slightly stale, significant gains can be obtain by using the state-based approach. Also, for clients that can tolerate some latency, as long as it is small, the best approach might depend on the size of the object.

Our conjecture is that benefits may be achieved by supporting *both* approaches simultaneously, such that for some realistic SLAs, an operation-based approach is used for larger objects and a state-base approach is used for smaller objects. Notice that the number of updates received by each object within a SLA time window also plays an important role deciding between approaches. To validate our hypothesis, we have developed a prototype of an hybrid system, that we have named Bendy, that is able to adapt the approach used for each object dynamically. This section reports on the design of this prototype and on the experimental results that validate our assumption.

A. Overview of the System

Bendy is, internally, composed of two independent SwiftCloud instances, which are hidden from the client by an extended SwiftCloud proxy.

One instance is the original, operation-based SwiftCloud implementation as described in [5]. Another instance is the version that we have produced that uses exclusively state-based CRDTs. Bendy makes dynamic decisions about which approach is more favourable for a given object, based on its size, on the associated SLA, and on the workload characterisation; the object state is stored in one of the instances accordingly. If, at some point, the implementation of an object needs to be changed in runtime, the state of the objects is transferred from one instance to the other, and the proxies updated such that the current implementation is used. Figure 1(a) show the interactions between the two instance with the proxy, and figure 1(b) represents the architecture what was described above.

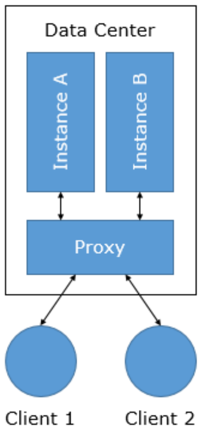
Although this approach is somehow rudimentary, and more efficient switchover between implementations could be supported by redesigning the metadata structure of SwiftCloud from scratch, we have opted not to do so for two reasons: first, the entire redesign of SwiftCloud is a challenge on its own, and outside the scope of this work; second, by using the current approach it is possible to make direct comparisons with the original implementation (i.e, all observed benefits derive from the use of dynamic adaptation, and not from other optimisations of the SwiftCloud middleware).

Note that the extended proxy maintains the metadata for both instances such that all accesses to a given object respect causality, regardless of the instance where the object is currently stored. Also, Bendy is focused only on the propagation of updates on the server side, and because of that, the remaining implementation of the clients is preserved from the original SwiftCloud system.

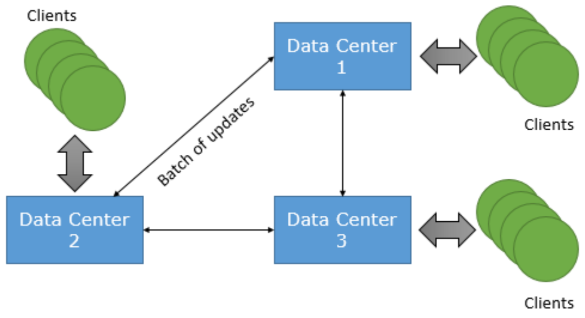
B. Taking SLAs Into Account

In Bendy, one SLA will be associated for all object. However, due to different access ratio of some objects, some updates are propagated as soon as possible, as in the original SwiftCloud implementation. The SLA is used mainly to decide when updates need to be propagated.

When deciding if an object should use an operation-based or a state-based approach, Bendy takes into account the global SLA, the object size, and the workload characterisation for that object, more precisely how many updates are expected to be performed during a period that



(a) Interaction between the two SwiftCloud instances and the proxy, representing a Data Center.



(b) Client view of the system

Figure 1. Overview of the described system.

corresponds to the SLA. If the ratio between the number of expected updates and the object size is above a given threshold, a state-based approach is used. Otherwise the default operation-based approach is used.

From the study that was made, we could observe that no significant advantages can be extracted from batching operation-based updates. Also, from the same study, we have seen that delaying operation-based updates may cause other updates to be stalled. Thus, for objects that are selected to use an operation-based approach, updates are propagated as soon as possible, exactly as in the original SwiftCloud implementation.

Regarding the state-based approach, one could observe in the study that significant gains can be obtained by delaying the propagation of updates. Therefore, in Bendy, we use the following strategy. Let δ_o be the latency tolerated for object o according to the corresponding SLA. Let μ_o be *propagation time*. Let t_o^u be the time at which some update u_o on object o has been performed at a given datacenter. To ensure that the object SLA is not violated, Bendy forces the propagation of that update (and all subsequent updates that have been buffered) at the following instant.

As it will be discussed in the next section, the propagation time μ_o depends not only on the size of the objects, but also on the number of updates that are propagated simultaneously. Thus, as discussed before, the value μ_o is adjusted in runtime, based on values measured during the last synchronizations.

C. Propagation Time

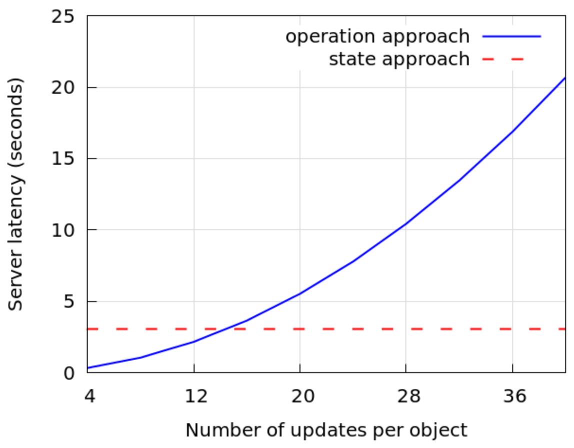
SLAs can be exploited to batch updates, a technique that can bring significant advantages for state-based approaches. However, batched updates need to be propagated *before* the SLA expires, with enough slack to transmit and deploy the updates at remote sites without violating the SLA. Therefore, the assessment of the time needed to propagate and apply updates, that we denote the *propagation time*, is of critical importance for any system that aims at exploring relaxed SLAs.

From our experience with SwiftCloud, we observed that the number of objects and the number of updates per object are the main variables that affect the propagation time. This is illustrated by the following experiments where we have fixed the SLA and object size and vary two other parameters, namely the number of objects and the number of updates batched: (i) in the first experiment, we fixed the number of objects to 200 and we vary the number of updates; (ii) in the second, we fixed the number of updates to 8 per object and we vary the number of objects.

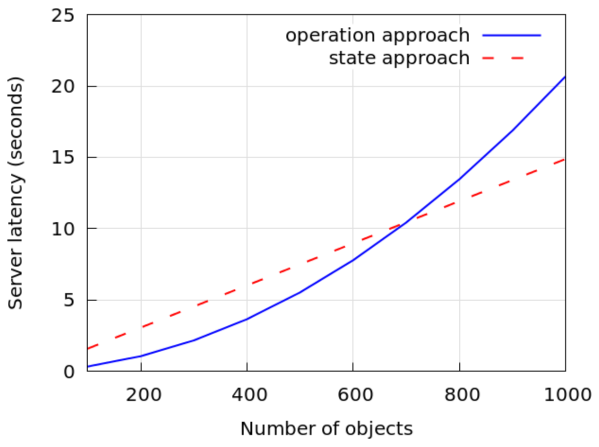
Figure 2(a) depicts the results from the first experiment. As expected, the number of batched updates are irrelevant in the state-based approach (as a single state per object is sent before the SLA expires). On the contrary, the propagation time increases significantly with the number of batched updates, in the operation-based approach. Interestingly, it can be observed that the propagation time grows super-linearly, because the system resources become exhausted in two many updates are queued to be applied at once. This further reinforces the observation that, at least in the current implementation of SwiftCloud, it is hard to extract benefits from relaxed SLAs with the operation-based approach.

Results for the second experiment are depicted in Figure 2(b). Unsurprisingly, the propagation time increases with the total number of updates but the interesting aspect is that the increase is linear with state-based approach while the operation-based approach depicts, as above, a super-linear degradation. However, these figures clearly show that, when computing the propagation time, one need to have a estimate of the total number of objects that are using batching, in order to compensate for the burst of updates that can result from having SLAs for multiple objects expiring simultaneously.

Since the parameters identified above, that influence the propagation time, are hard or impossible to predict offline, in Bendy we keep in runtime statistics for the propagation time for the subset of the top-k objects that use a state-based approach.



(a) Updates per object vs Latency on server side



(b) Number of objects vs Latency on server side

Figure 2. Propagation Time

In this prototype, the propagation time t_x^i in round i, for each of the objects O_x that use state-based, is computed dynamically as follows:

$$t_x^i = t_{u_x}^i + \delta - \mu_x^i$$

where $t_{u_x}^i$ is the time at which O_x received the first update operation u_x for this round; δ is the visibility delay tolerated by the application (specified in the SLA) and; μ_X^i is a moving average over a window of the time that the last n coordination procedures took.

D. Dynamic Adaptation

Recent work on adaptive storage systems[16] provides evidence that substantial gains can be achieved without performing fine-grain adaptation of every and single object in a storage system. In fact, many realistic workloads follow a zipfian distribution, where some objects are accessed much more often than the others. Thus, a large fraction of the

Algorithm III.1 Process of migrating objects from one instance to the other.

```
function MIGRATE
    $O$ :=GETOLDTOPK()
    $E$ :=GETNEWTOPK()
    $N := (O \setminus E) \cup (E \setminus O)$    ▷ $N$ = all objects to be migrated
    while $N \neq \emptyset$ do
        $m \subseteq N$
        LOCKOBJECTS($m$)
        MIGRATEOBJECTS($m$)
        UNLOCKOBJECTS($m$)
        $N := N \setminus m$
function MIGRATEOBJECTS($m$)
    for $id$ in $m$ do
        $o$ :=FETCHOBJECT($id, replicaA$)
        $o$ :=RECONFIGURE($o$)    ▷ Each type of object implements its own mechanisms to change from one approach to the other
        STOREOBJECT($id, o, replicaB$)
function LOCKOBJECTS($m$)
    for $id$ in $m$ do
        LOCKCLIENTACCESS($id$)
    FLUSHPENDINGUPDATES()
function UNLOCKOBJECTS($m$)
    FLUSHPENDINGUPDATES()
    for $id$ in $m$ do
        UNLOCKCLIENTACCESS($id$)
```

gains can be achieved by adapting only the implementation of those popular objects.

1) Reconfiguration: Based on these observations, Bendy performs a top-k analysis of the workload and only adapts the implementation of the most popular object. All other objects just use the default SwiftCloud implementation. The state-of-the-art stream analysis algorithm [17] permits to infer the top-k most frequent items of a stream in an approximate, but very efficient manner. Given that workloads may change in run-time, the top-k analysis is repeated periodically. The Algorithm III.1 describes the whole process. At the end of each period, Bendy first reverts back to the default (operation-based) implementation all objects that are no longer part of the top-k. For those objects in the top-k. Bendy selects the target implementation using the criteria described above. Finally, Bendy reconfigures the implementation of those objects for which the target implementation differs from the current implementation in use.

Given that Bendy is implemented as a wrapper, the reconfiguration of a given object is implemented by migrating that object from one SwiftCloud instance to the other. In this process, N objects are migrated, and to ensure consistency, we lock the access to an object that is being reconfigured to the other implementation. The action of locking access to an object might have impact on the throughput, specially if we lock the access to all the objects, that are being migrated, at the same time. To avoid such drawback, the system can migrate m objects at a time until all N objects are migrated. Being m a fraction of N. More details are provided in section IV-D with experiments that compare different values for m.

2) Selecting the Right Implementation: Bendy requires several statistics about the objects and the workload to be maintained, such as the object size, the update ratio, and the time it takes to propagate and apply state updates. To keep those statistic for every object in the storage system may cause an unnecessary overhead. Also, client proxies need to be aware of which instance stores a given object.

When deciding if an object should use an operation-based or a state-based approach, Bendy takes into account the SLA, the object size, and the workload characterisation for that object, more precisely how many updates are expected to be performed during a period that corresponds to the SLA. If the ratio between the number of expected updates and the object size is above a given threshold, a state-based approach is used. Otherwise the default operation-based approach is used.

E. Implementation Issues

During the development of the system, many decisions were made to implement the features that were described above. The main features are: the implementation of state approach, the combination of the two approaches operation- and state-based and the dynamic system.

1) Implementation of the state-based approach: As it was discussed before, we started our project with an implementation of SwiftCloud that only supports operation-based CRDTs. The first step was to implement a state-based solution to make the first comparisons between the two approaches. Thus, we based our implementation on the original algorithms from [2] and managed to implement the merge function. For that, we've extend the CRDT interface and changed the internal representation of the objects like in the algorithms. However, we've only changed how the updates where propagated on the server side, because we assumed that one client will never make enough updates that justifies the sending of a state. Instead, we look at the client as an entity that makes operations remotely and not as a replica, like in the original SwiftCloud. We've kept all the features of the Client and focused only on the server side.

The second part of implementing the state-based approach was the propagation of updates. At the same time we did the propagation of updates we started to add the notion of SLA. Since, in the original SwiftCloud, the updates were sent asynchronously, it was not hard to implement the SLA. Basically we've added a delay between synchronizations to the synchronization thread. Later, we've implemented the formula described in Section III-C to ensure that the SLA was satisfied. About the propagation of updates, we had to implement the merge of the replica's clock to ensure that clients were able to see updates from other replicas. Although in the specification of the state approach it is not required to have a causal clock outside of the object, we've found out that it was harder change completely the SwiftCloud implementation than keeping the clock. Besides that, we intended to integrate with the operation-based approach which needs that global clock.

2) Combining operation-based and state-based approaches: After the implementation of the state approach, we tried to put together both approaches in the instance. However, that was impossible due to the complexity of the global causal clock and because, when merging an object, the merge of the clock sometimes it contain the timestamps of other operations. This means that after making a merge of an object and its clock, the operations that were on hold to be applied will be discarded because the new clock already includes the timestamp of those operations. We've looked at this problem for a long time, and the best solution was to separate the two implementations into two instances. Otherwise, we wouldn't have time to work on the dynamic part of the system.

To hide the two instances from the client we created a proxy. This proxy basically forwards the client's request to the instance that has the object. For the client there were no changes. We kept the same interface so that the client can make operations regardless of the propagation method that each instance uses.

3) Implementing a dynamic system: Regarding this topic, most of the details are explained in the Section III-D. Nevertheless, there is one aspect that needs to be explained with more detail. Regarding the migration of an object from one approach to the other, we had to make it in a simple manner. Because we have two instances, the causal clocks of each instance will grow independently. However, we need to move one object from one side to the other and changing the clocks was not easy. The best solution was to momentarily lock the object on the two instances for all clients except one. There is one client that belongs to the system and it is in charge of migrate objects. This client, contacts directly the instance instead of contacting the proxy like the other clients. It reads the value on the original instance, removes the object from that instance (by removing its the elements in the case of the set) and applies that value on the other instance. Finally we flush all buffers to ensure that all replicas have migrated the object and we unlock the

IV. Evaluation

This section presents an evaluation of Bendy. We first present our experimental setting, where we list the systems we use to compare Bendy with. Then we present experiments comparing the throughput and the bandwidth used by each of the systems. We finally evaluate how Bendy is able to adapt to changes in the workload.

A. Experimental Setup

Each experiment uses 3 datacenters. Each datacenter replicates the full key-space which is composed by 1000 objects for the experiments in Subsections IV-B and IV-C, and 50000 objects for the experiments in subsection IV-D. We run a total 600 clients, having 200 client associated with each datacenter. In order to generate the workload, we use a zipfian distribution. All objects have associated a SLA of $10s$. We first run a warm-up phase were the database is populated. Then, each experiment runs for 5 minutes.

In our experiments, we compare the following three systems:

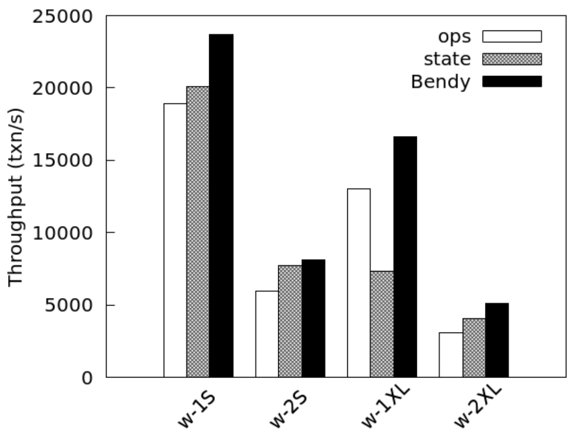
- An unmodified version of SwiftCloud (**op-based** hereafter) that implements an operation-based dissemination process. Updates are propagated immediately without batching them.
- A modified version of SwiftCloud (**state-based** hereafter) that implements a state-based dissemination process. Updates are batched based on the formula presented in Subsection III-C.
- Bendy which is a mixed approach that follows the specifications presented in Section III. It is approximately 2200 lines of Java. Bendy is built on top of SwiftCloud.

B. Throughput

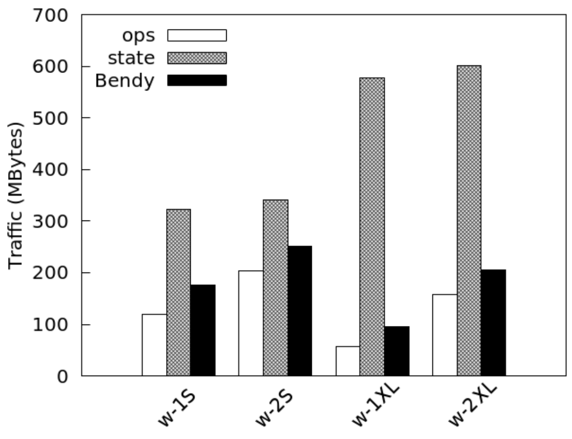
In the following experiment we aim at comparing the three systems in terms of throughput. Since Bendy runs two independent instances of SwiftCloud, one implementing operation-based and other one implementing state-based, bridged by an extended proxy, we opted for running the experiments in two instances also for the other systems. This allows us to achieve comparable results; even though op-based and state-based systems could run the whole experiment using a single instance since they do not mix approaches.

Figure 3(a) show the results for two different workloads: (i) a read-dominated workload with only 5% of updates (Workload-1); and (ii) a balanced workload with an equal number of reads and updates (Workload-2). These are two of the workloads specified by the widely used YCSB[18] benchmark and are representative of the workloads imposed by typical applications of geo-replicated storage systems. In addition, for each of the workloads, we experiment with two different object sizes: S (12KB) and XL (30KB). Bendy, due to the parametrization of the zipfian distribution generator and our top-k analysis, optimizes the dissemination process of 1% of the total number of objects. In these experiments, we are forced to limit the number of objects to 1000 since the state-based approach starts struggling and becoming very unstable with a large number and size of objects, due to the amount of information that needs to ship in every *coordination procedure*. Operation-based and Bendy do not suffer from this problem but in order to have comparable results across systems we limit the number of objects also for them. In the experiment in Subsection IV-D we use a more realistic amount of objects (50000) to demonstrate that our system does not suffer from this problem.

The results show that Bendy always outperforms the other two systems (up to 127% in some cases). The reason is because our system does not reach the bottleneck stage that the other solutions have to face. The state-based solution has to struggle with large-sized objects while the operation-based approach struggles with the high amount of operations



(a) Throughput



(b) Bandwidth

Figure 3. Throughput vs Bandwidth

issued in a few objects. This experiment validates our hypothesis, and demonstrates that benefits can be achieved by having a hybrid system.

In this experiment, the state-based approach seems to, most of the times, outperform the operation-based approach. However, for the state-based approach, the SLA of $10s$ was only satisfied once as we can see in the Table I. The main reason for not being able to satisfy the SLA was the large amount of states that are processed at the same time and its size. Nevertheless, as our study unveils, this directly depends on the workload, the SLA, and the size of the objects.

	ops	state	Bendy
Workload-1S	1 523	11 640	267
Workload-2S	377	5 290	237
Workload-1XL	1 238	55 119	799
Workload-2XL	492	63 120	729

Table I
COORDINATION TIME (AVERAGE)

Finally, one more subtle conclusion one can extract from this experiment is that, with the state-based approach, one must take special care to avoid violating the SLA. We notice that, as soon as the size and the number of objects increases, the accumulated processing time for all state-updates also increase. Thus, a naively configured state-based approach can easily start violating the target SLA. For instance, Table I lists the average time that the *coordination procedures* took for each of the experiments. One can see that the state-based solution is only able to satisfy the SLA of $10s$ for the one of the experiments (Workload-2S), violating in the other three, e.g. for Workload-2XL, it takes more than $60s$ on average. This reinforces the importance of applying the state-based approach just to a small number of top-k objects, that have the bigger impact on the system performance.

C. Bandwidth Utilization

With the same experiment, we also measured the bandwidth usage of the different approaches. Figure 3(b) shows the amount of bandwidth, in MB, used by each of the approaches for the entire run of five minutes. The experiment configuration and the used workloads are equivalent to the ones presented in the previous subsection.

The results match our analysis. As expected, the pure state-based system, is by far the worst solution. In many cases this system sends the state of objects that only received 2 or 3 operations, which is not efficient. Regarding Bendy, it uses more bandwidth than the pure operation-based system. Nevertheless, as demonstrated before, we reach better throughput mostly because of (i) the benefit of batching updates, and (ii) the inter-object dependencies problem of the operation-based system described in our study.

D. Dynamic Behavior

In previous experiments, we have compared the throughput of all the approaches at a stable point. However, in a real setting, the workload may change dynamically. Therefore, in this subsection, we describe an experiment where we induce a dynamic change in the workload by changing the most accessed objects. Our goal is to assess how well Bendy adapts to the changes and how the adaptation penalizes the throughput provided by Bendy.

For this experiment, we use a balanced workload with an equal number of reads and updates. The experiment goes as follows: during the first 100 seconds, the system is stable, meaning that no changes in workload are introduced; then the most accessed objects are changed. This forces Bendy to migrate a total of 700 objects between instances in order to optimize the newly identified top-k objects. We compare four variations of Bendy:

1) A version that does not adapt to the new workload (*baseline*);
2) A version that migrates all objects at once (*All at once)*;
3) A version that migrates the objects in groups of 50 (*50 by 50*);

4) A version that migrates the objects in groups of 10 (*10 by 10*).

Figure 4 shows the throughput of each of the variations. One can see that, after changing the workload (100^{th} second), all variations of Bendy degrade their performance reducing its throughput almost 33%. This is because the objects being optimized are not highly accessed anymore and the new hot objects are using basic operation-based approach, which does not benefit from batching. Then, after 125 more seconds (225^{th} second), the migration process is started. As expected, if we move all objects at once the throughput drops drastically until the balancing process ends. However, if we migrate a small amount of objects at a time, the process may take longer but the throughput loss is minimal. We can conclude that moving by groups of around 50 objects is a reasonable solution. Although the throughput initially drops (about 20%), it recovers quite fast, achieving maximum throughput in less than 50 seconds since the balancing process started.

Another aspect of our dynamic system is the ability to detect that the top-k list has changed. As in all autonomic system, there is a tradeoff between how fast the system reacts to changes and the likelihood of the new state to be stable. Since we consider this problem orthogonal to this thesis, we decided to adopt a simple approach, in which we wait for some extra time once the change has been detected, before starting the adaptations procedure of Bendy. Of course, we could adopt more sophisticated techniques in order to make Bendy more robust to transient workload oscillation, such as techniques to filter out outliers [19], detect statistically relevant shifts of system's metrics [20], or predict future workload trends [21].

In order to get some insights on the time we have to wait until considering a change in the workload stable, we evaluate the time that our top-k analysis needs in order to propose a top-k list close to the optimal. For the implementation of top-k that was used, there is a variable called *capacity* which is the number of events that happened in the past. In our solution, each event is an access to an object. This means, if the algorithm uses a large capacity then we will have more accuracy in the result. However, if the capacity is extremely large, we will waste a lot of resources in terms of memory and it should take more time to process a list of top-k objects. Figure 5 shows the results of the experiment. As one can see, a larger capacity brings more accuracy, and the top-k analysis rapidly starts proposing almost an optimal list of hot objects (only 10% of error) in barely 75 seconds. This justifies the 125 seconds time window used in the previous experiment before adapting the system.

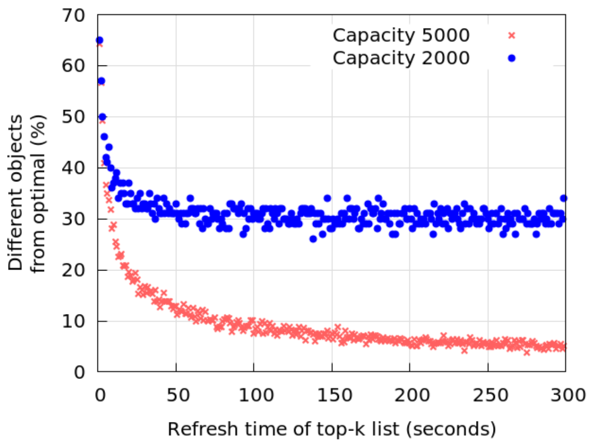


Figure 5. Percentage of objects in a top-k list that are different from the optimal top-k list.

E. Discussion

The results obtained with Bendy, clearly show that significant benefits can be achieved if multiple CRDT implementations are supported in a single system, and the best implementation is used according to the user SLA and the workload characterization. This opens the door for new avenues of research, in the design and implementation of systems that can provide such functionality in an much more integrated manner, than that provided by the wrapper approach followed in the Bendy design. In the following paragraphs, we discuss a number of insights and guidelines, that we gained from our experience with the evaluation SwiftCloud and the implementation of Bendy, that may be useful when building future systems:

- If updates need to be pushed as soon as possible, the current SwiftCloud operation-based approach excels. However, if clients can tolerate stale data, significant gains could be achieved by supporting state-based propagation of updates.
- With the current SwiftCloud implementation, no significant advantages can be extracted from batching multiple operation-based updates. This happens because SwiftCloud applies all the batched operations serially and independently (for instance, releasing and grabbing locks for every single operation in the batch). There is room to optimize SwiftCloud for more efficient processing of batched updates. Also, semantic compression of batched updates, as suggested by in [9], would also improve the system.
- The way SwiftCloud compresses client metadata, namely by using a single clock shared by all objects, makes hard, if not impossible, to support multiple distinct SLAs in an effective manner, as the interdependencies that are created among updates may cause SLAs to be violated. Techniques that are more costly, but that allow for more fine-grain tracking of dependencies (such as using per-object clocks [22]) are needed to effective support multiple SLAs.

Taking into consideration our experience, implementations that aim at outperforming Bendy should use the following guidelines:

- To rely on CRDT implementations that support both approaches, like the Optimized OR-Set [15].
- To use metadata maintenance techniques that avoid cre-

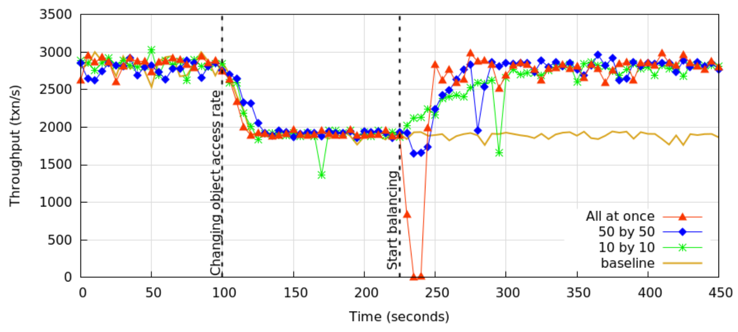


Figure 4. Throughput of one DC during the balancing process of top-k objects.

ating false dependencies among objects using different SLAs.

- To use metadata maintenance techniques that ensure that causality information regarding objects using different implementations is not compressed together, as this creates undesirable dependencies among both implementations.
- Embed in the transaction processing engine sensors that may simplify the task of extracting the workload characterisation, namely the update rate, given that the benefits of the state-based over the operation-based critically depend on the possibility of aggregating multiple updates before the SLA expires.

V. Conclusions

In this thesis we have analyzed the cost performance tradeoffs between the two main approaches that can be used to propagate updates in geo-replicated stores using CRDTs, namely operation-based and state-base approaches. Our work shows that node of the approaches outperforms the other in absolute terms, and that an hybrid system may yield the best results. In particular, we have show that, if the application is willing to tolerate small amounts of staleness when reading objects, significant throughput gains can be achieved by encoding multiple updates in a single state-update.

In order to validate our hypothesis, we have presented and evaluated Bendy, a CRDT-based geo-replicated storage system that supports both operation- and state-based approches. Bendy is able to optimize object-wise for a bounded number of objects (hot objects) the dissemination process. Plus, Bendy is able to react to changes in the workload by relying on an approximate, but very efficient, state-of-the-art stream analysis algorithm. Our results have shown that Bendy outperforms solutions that use only one of the two approaches, and that is capable of rapidly self-adapt to variations in the workload.

In the process of implementing and experimenting with both approaches, using SwiftCloud, a state of the art CRDT-based geo-replicated storage system, we were also able to get interesting insights that may help in driving future implementations of similar systems. In particular, designers need to take special care on providing support for efficient processing of batched operation-based updates, fast state-updates, and carefully crafted metadata structures that avoid undesirable false causal dependencies among objects.

Acknowledgments

I would specially like to thank Manuel Bravo for the fruitful discussions and his helpful comments, he was constant source of guidance and support on every stage of the research work. And for his contribution on the paper produced by this work.

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/2013, via the project PEPITA (PTDC/EEI-SCR/2776/2012)

References

[1] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050613.2050642

[4] M. Letia, N. M. Preguiça, and M. Shapiro, "Crdts: Consistency without concurrency control," *CoRR*, vol. abs/0907.0929, 2009. [Online]. Available: http://arxiv.org/abs/0907.0929

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

[6] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro, "Incremental stream processing using computational conflict-free replicated data types," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, ser. CloudDP '13. New York, NY, USA: ACM, 2013, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/2460756.2460762

[7] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 395–403. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2009.20

[8] C. Baquero, P. Almeida, and A. Shoker, "Making operation-based crdts operation-based," in *Distributed Applications and Interoperable Systems*, ser. Lecture Notes in Computer Science, K. Magoutis and P. Pietzuch, Eds. Springer Berlin Heidelberg, 2014, pp. 126–140. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-43352-2_11

[9] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by delta-mutation," *CoRR*, vol. abs/1410.2803, 2014. [Online]. Available: http://arxiv.org/abs/1410.2803

[10] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage." in *Proceedings ACM Symposium on Operating Systems Principles*. ACM, November 2013. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=201390

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[12] R. Klophaus, "Riak core: Building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*, ser. CUFP '10. New York, NY, USA: ACM, 2010, pp. 14:1–14:1. [Online]. Available: http://doi.acm.org/10.1145/1900160.1900176

[13] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, ser. PDIS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 140–149. [Online]. Available: http://dl.acm.org/citation.cfm?id=645792.668302

[14] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[15] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte, "An optimized conflict-free replicated set," *CoRR*, vol. abs/1210.3368, 2012. [Online]. Available: http://arxiv.org/abs/1210.3368

[16] M. Couceiro, G. Chandrasekara, M. Bravo, M. Hiltunen, P. Romano, and L. Rodrigues, "Q-opt: Self-tuning quorum system for strongly consistent software defined storage," in *Proceedings of the 16th International Middleware Conference*, ser. Middleware '15. New York, NY, USA: ACM.

[17] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Database Theory - ICDT 2005*, ser. Lecture Notes in Computer Science, T. Eiter and L. Libkin, Eds. Springer Berlin Heidelberg, 2005, vol. 3363, pp. 398–412. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30570-5_27

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[19] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, 2004. [Online]. Available: http://dx.doi.org/10.1007/s10462-004-4304-y

[20] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, no. 1/2, pp. pp. 100–115, 1954. [Online]. Available: http://www.jstor.org/stable/2333009

[21] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Fluids Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 401–416. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043593