Dynamic Adaptation of Geo-Replicated CRDTs*

Carlos Bartolomeu†, Manuel Bravo†‡, Luís Rodrigues†
†INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
‡Université Catholique de Louvain, Belgium
{carlos.bartolomeu, angel.bravo@uclouvain.be, ler}@tecnico.ulisboa.pt

ABSTRACT

Conflict-free Replicated Data Types (CRDTs) are high-level data types that can be replicated with minimal coordination among replicas due to its confluent semantics. This property makes CRDTs specially appealing for geo-replicated settings. Different approaches, such as state transfer and operation forwarding, have been proposed to propagate updates among replicas, with different tradeoffs among the amount of network traffic generated and the staleness of local information. This paper 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.

CCS Concepts

•**Computer systems organization** → **Architectures; Self-organizing autonomic computing;** *Distributed architectures;* •**Information systems** → *Key-value stores;*

Keywords

CRDTs, Operation-based, State-based, Geo-replication

1. INTRODUCTION

With the advent of cloud computing, and the need to maintain data replicated in geographically remote datacenters, the quest for strategies able to provide data consistency with minimal synchronization became of utmost practical relevance. Unfortunately, most data types require operations to be totally ordered to ensure replica consistency. This means that either i) operations are diverted to a single primary replica, incurring on long delays and availability problems, or ii) that expensive consensus protocols, such as Paxos [8], are used to order the updates.

Conflict-free Replicated Data Types (CRDTs) [9, 15, 16] are data types whose concurrent operations do not conflict with each other and, therefore, can be replicated with minimal coordination among replicas. CRDTs are designed in such a way that any two concurrent operations are commutative and, therefore, even if they are executed in different sequential orders, the same final result is reached. As a result, replicas can locally execute operations promptly, without prior synchronization with other replicas, and later ship these operations to other replicas when it is convenient. Using this approach, even if replicas diverge temporarily from each other, convergence is eventually reached due to the commutative property. Thus, unlike other eventual consistency approaches, CRDTs strongly simplify the development of distributed application such as social networks, collaborative documents, and online stores [12, 14, 20].

Two main types of CRDT implementations have been proposed, which differ on the approach used to reach eventual consistency, namely *operation-based* [2, 15] and *state-based* [15] CRDTs. Operation-based CRDTs ship to the remaining replicas the operations that are executed locally, which are eventually remotely executed. 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 in the following sections. In fact, depending on the scenario, one approach may be more suitable than the other.

In this work, we study the state- vs operation-based tradeoffs in CRDTs. As a result, we highlight some interesting insights on the implementations of CRDT-based storage systems. Furthermore, to address the challenges in choosing among CRDT implementations, we propose BENDY, a self-adaptive version of SwiftCloud [20]. BENDY is able to achieve better performance than: (i) systems that only support one implementation, and (ii) systems that support multiple implementations but do not self-adapt and, instead,

*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), by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement nº 609551, and by the Erasmus Mundus Joint Doctorate Programme under Grant Agreement 2012-0030.

SAC 2016, April 04-08, 2016, Pisa, Italy
© 2016 ACM. ISBN 978-1-4503-3739-7/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851641

rely on a static configuration. The main contributions of this paper can be enumerated as follows:

• We present a detailed analysis of the tradeoffs posed by the use of different CRDT implementations (Sec. 3). Namely, we compare the state- and operation-based approaches, which are the most relevant implementations proposed in the literature. For this purpose, we have implemented a variant of SwiftCloud that implements a state-based approach.

• We present the design and implementation of BENDY, a self-adaptive CRDT data store that automatically selects the best CRDT implementation for each of the most heavily used objects in the store (Sec. 4). BENDY is able to dynamically change an object CRDT implementation as the features of the objects and of the workload vary in runtime. By performing fine-grain adaptation only on the most accessed objects, BENDY avoids the scalability limitations that would result from monitoring all objects in the store.

• We present results from an experimental evaluation of BENDY (Sec. 5). Our results show that BENDY is capable of achieving 41% better throughput in average in comparison to both the original SwiftCloud and the modified version of SwiftCloud exclusively supporting state-based CRDTs.

• We discuss insights, extracted from our experience building BENDY, on critical aspects that need to be considered when building a SwiftCloud/BENDY-like data store (Sec. 6).

2. BACKGROUND

In the following section, we describe the main CRDTs approaches proposed in the literature, namely operation- and state-based CRDTs. We also describe SwiftCloud [20], a state-of-the-art geo-replicated data store that illustrates how CRDTs can be used in practice.

2.1 CRDT implementations

Conflict-free Replicated Data Types (CRDTs) are data structures that allow replicas to be updated concurrently and still ensure that all replicas eventually converge to the same state [15, 16]. CRDTs are a powerful alternative to simpler approaches to reconcile divergent replicas, such as user-specified functions [17], which make programming hard, or last-write-wins semantics [19], 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 [3]).

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.

2.2 SwiftCloud

SwiftCloud is a geo-replicated cloud storage system that stores CRDTs and caches data at clients [20]. 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.

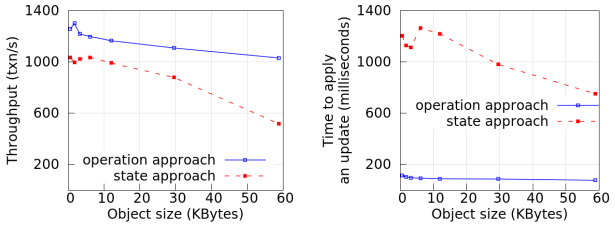
The way transactions are propagated and applied provides causal+ consistency [10]. Causal consistency is tracked and enforced by relying on a vector clock with an entry per datacenter. SwiftCloud implementation is based on a per-datacenter serialization point that sequences local updates and enforces causal dependencies.

3. STATE- VS OPERATION-BASED CRDTS

We now study the tradeoffs between the use of state- and operation-based CRDTs in geo-replicated settings.

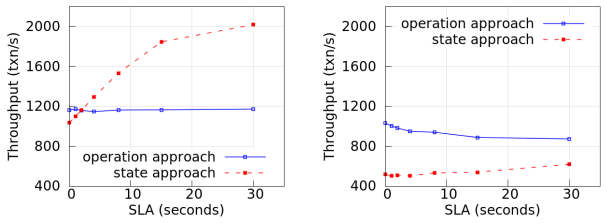
3.1 Eager Propagation

SwiftCloud uses an operation-based approach because, in settings where one tries to maximize data freshness, by propagating updates immediately, an operation-based implementation is expected to provide better performance than a state-based implementation. We have collected data that supports this design choice. Figure 1 shows the impact of the object size when propagating updates among two datacenters using both operation- (continuous line) and state-based approaches (dotted line). Figure 1(a) shows the system throughput, measured in number of transactions per second and, finally, Figure 1(b) shows the time taken by operations to be applied in the preferred datacenter. Unsurprising, these results indicate that, in this setting, the operation-based approach always offers the best performance, both in



(a) Throughput (b) Latency

Figure 1: Impact of the object size for both approaches



(a) Small object size (12KB) (b) Large object size (59KB)

Figure 2: Relaxing data freshness requirements

terms of throughput and latency. As expected, the operation-based approach does not get significantly affected by the object size since the size of the data propagated across datacenters is reasonably constant independently of the object size. This does not hold in the state-based solution which penalizes this approach.

3.2 Supporting Different SLAs

In a real setting, different applications may have different requirements in terms of data freshness. Examples of applications that can tolerate slightly stale data are web search applications, social networks, web-based e-mail, calendaring programs, news readers, personal cloud file systems, and photo sharing sites, among others [18]. Typically, these and other requirements (such as latency guarantees) are captured by Service-Level-Agreements (SLAs hereafter). Introducing relaxed requirements brings the possibility of batching multiple updates. It is therefore relevant to explore whether gains can be achieved by using a state-based approach when different SLAs can to be supported. For that purpose we did a number of experiments whose results we report below. All these experiments use a workload of 50% reads and 50% writes, and a duration of 5 minutes. The number of clients varies between 5 and 20 clients per datacenter depending on the specific experiment, with the aim of reaching the maximum system capacity.

We study the impact of buffering updates and delaying its propagation to other datacenters. In this experiment, we fix the object size and vary the SLA. The impact of the SLA on the operation-based approach is that multiples updates can be batched and sent in a single message to other datacenters. This may yield some improvement on the network utilisation. For the state-based approach, this means that a single state-update is sent instead of multiple operations. The goal of this experiment is to understand, for different object sizes, how many updates need to be buffered to compensate the additional overhead of sending the entire state. The intuition is that the weaker the requirements of the application (i.e. more updates can be batched), the better the state-based approach performs.

Figure 2 shows the results for a database of objects with fixed size and a deployment of three datacenters. Figure 2(a) uses a fixed size of 12KB while Figure 2(b) uses a fixed size of 59KB. The x axis of both plots show different freshness of data requirements, the larger the SLA constraint, the more relaxed the SLA is. The initial value of SLA=0 is equivalent to the original SwiftCloud implementation without any batching. The y axis show the system throughput. As in previous experiment we compare both operation- and state-based approaches.

Interestingly, for small objects (Figure 2(a)), even for very small relaxations of the SLA, the state-based approach outperforms the operation-based approach. In fact, significant throughput improvements can be achieved if the application can tolerate a certain amount of staleness in the observed data. For instance, for a SLA of $15s$ the throughput of the system can be 50% higher if the state-based approach is used. On the other hand, for large objects, as the tendency showen in Figure 2(b) implies, state-based may only outperforms the operation-based approach for quite large SLAs. Thus, deciding whether to use state- or operation-based approach not only depends on the SLA size but also in the size of the objects being stored.

An additional conclusion extracted from this experiment is that no advantages can be obtained from SLAs with the operation-based approach. Although the relaxed SLA allows to batch multiple updates in a single message, the fact that all operations need to be applied serially (due to the SwiftCloud design) via a computationally expensive procedure, counterbalances this potential benefit. One potential avenue for research, that has not been explored in this work, is to find techniques to merge operation-based updates based on semantic information, and combine this with techniques to apply updates in parallel.

3.3 Impact of False Dependencies

Finally, we have performed experiments to understand the behavior of SwiftCloud when different objects have different SLAs. In particular, we are interested in assessing if metadata management in SwiftCloud, namely the use of a single clock per datacenter to enforce causal consistency, can be an impairment when trying to ensure bounded staleness.

More precisely, we consider a SwiftCloud system with two different sets of clients. The first set of clients accesses exclusively object O_1, which has a SLA of $5s$ (i.e., clients tolerate reading data that is $5s$ stale). The second set of clients accesses exclusively object O_2, which has an $8s$ SLA. In this experiment, we only deploy two datacenters.

Each spot (x, y) in Figure 3 represents the time y that took to the remote datacenter to process the batched updates sent after batching for x time in the local datacenter. Due

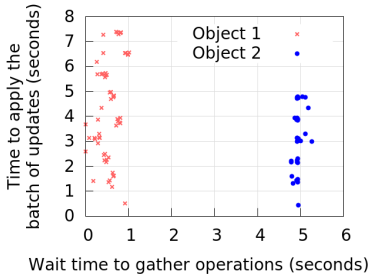


Figure 3: Time required by a remote datacenter to process a batch of operations propagated from another datacenter.

to the experiment setup, no real dependencies across objects are introduced by the clients. Therefore, the processing time should be quite stable. However, SwiftCloud's design decision of having a single clock per datacenter implies that all operations issued in a datacenter are totally ordered. Thus, due to this metadata compression, a large amount of false dependencies is created among clients of different objects. This induces interference among the otherwise independent streams of request: although the size of the operations batch is approximately the same, the time used to process the operations from a given batch in the remote datacenter varies from $0.5s$ to $7.5s$ for O_1, and from $0.5s$ to $5s$ for O_2. This experiment shows that it may be unfeasible to support clients with different SLAs with the current SwiftCloud implementation. Note that this problem would not manifest if objects share the same SLA. In such a scenario, if update u_2 is issued after update u_1 then, since u_1's SLA expires first, u_1 will be necessarily applied before u_2, without obstructing u_2, even if the system creates a false dependency between both updates.

4. BENDY, A HYBRID SOLUTION

The experiments above suggest 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 when choosing between approaches. To validate our hypothesis, we have developed a prototype of a 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.

4.1 Architecture

SwiftCloud design makes extremely hard to support simultaneously state-based and operation-based implementations for different objects, mostly due to their metadata compression technique (Section 3). Instead of attempting to redesign SwiftCloud from scratch, we have opted to implement BENDY as a wrapper that encapsulates 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 [20]. 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.

This design choice makes adaptation a bit slower than a more integrated design, where both operation- and state-based CRDTs would be supported simultaneously by the same instance. On the other hand, for the purpose of this study, it has the important advantage of allowing for a direct comparison with the original SwiftCloud system, which is left unchanged. This allows to decouple the effects of dynamic adaptation from the effects of other changes to the SwiftCloud middleware, that would be required to support a more integrated approach.

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 clients is preserved from the original SwiftCloud system.

4.2 Selecting the Right Implementation

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, particularly 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. More precisely, BENDY uses state-based approach for an object O_x if the following holds:

$$U_x^{SLA} * size(u_x) > size(O_x)$$

where U_x^{SLA} is the number of updates received for object O_x during the time window of a given SLA, $size(u_x)$ is the average size of an object O_x update, and $size(u_x)$ is the average size of object O_x's state.

4.3 Dynamic Adaptation

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. Again, to maintain and update such directory information for every single object may be impracticable.

Fortunately, recent work on adaptive storage systems [5] has shown 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

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

Based on these observations, BENDY performs a top-k analysis of the workload and only adapts the implementation of the most popular objects. All other objects just use the default SwiftCloud implementation. BENDY uses a state-of-the-art stream analysis algorithm [11] that 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. 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.

4.4 Reconfiguration

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 our current prototype, migration uses a simple "stop-and-go" strategy: the access to the object is temporarily locked and a state-synchronisation is forced among replicas (by pushing all pending updates). When all replicas of the object reach a quiescent state, its state is transferred from one instance to the other and, subsequently, the object is unlocked.

If, at the end of each period, the number of objects to be reconfigured is large, BENDY performs the reconfiguration in multiple steps, where in each step only a fraction of the objects is reconfigured. This mitigates the negative impact that the reconfiguration may have in the system throughput. More details on this strategy are provided in Subsection 5.4.

4.5 Propagation Time

As discussed before, 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. Unfortunately, the exact quantification of this impact is hard to perform analytically and/or offline. Therefore, in BENDY we opted to keep in runtime statistics for the propagation time for the subset of the top-k objects that use a state-based approach, and adapt the propagation time every *synchronization round*. A synchronization round is delimited by the following coordination procedures: (i) the propagation of the updates, encoded in the state; and (ii) their deployment in the remote datacenter.

In our current 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.

5. 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.

5.1 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 5.2 and 5.3, and 50000 objects for the experiments in subsection 5.4. We run a total of 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 $10s$. For the experiments, we first populate the database; then, each experiment runs for 5 minutes.

The experimental test-bed used is a private cloud composed by a set of virtual machines deployed over 20 physical machines (8 cores and 40 GB of RAM) connected via a Gigabit switch. Each datacenter is simulated by a single VM. We also use separate VMs for each 100 clients. Each VM, which runs Ubuntu 14.04, and it is equipped with 8 (virtual) cores, 10GB disk and 35GB of RAM memory, is allocated in different physical machines.

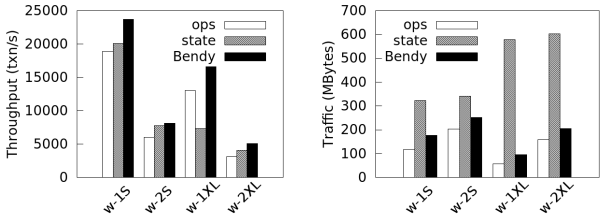
In our experiments, we compare the following three systems:

• 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 4.5. This version takes advantage of the state-based relaxed delivery requirements and it circumvents the limitations originated by the causal and exactly-once delivery requirements of the operation-based approach.

• BENDY which is a mixed approach that follows the specifications presented in Section 4. It is approximately 2200 lines of Java. BENDY is built on top of SwiftCloud.

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



(a) Throughput (b) Bandwidth

Figure 4: Throughput vs Bandwidth

run the whole experiment using a single instance since they do not mix approaches.

5.2 Throughput

In the following experiment we aim at comparing the three systems in terms of throughput.

Figure 4(a) shows 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 [4] 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 it becomes 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 5.4 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 issued in a few objects. This experiment validates our hypothesis, and demonstrates that benefits can be achieved by having a hybrid system.

	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 1: Coordination time (in *ms*) on 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 1 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 10*s* for one of the experiments (Workload-2S), violating in the other three, e.g. for Workload-2XL, it takes more than 60*s* 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.

5.3 Bandwidth Utilization

With the same experiment, we have also measured the bandwidth usage of the different approaches. Figure 4(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 described in the previous subsection.

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 Subsection 3.3.

5.4 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: (i) a version that does not adapt to the new workload (*baseline*); (ii) a version that migrates all objects at once (*All at once)*; (iii) a version that migrates the objects in groups of 50 (*50 by 50*); and a version that migrates the objects in groups of 10 (*10 by 10*). Figure 5 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 be-

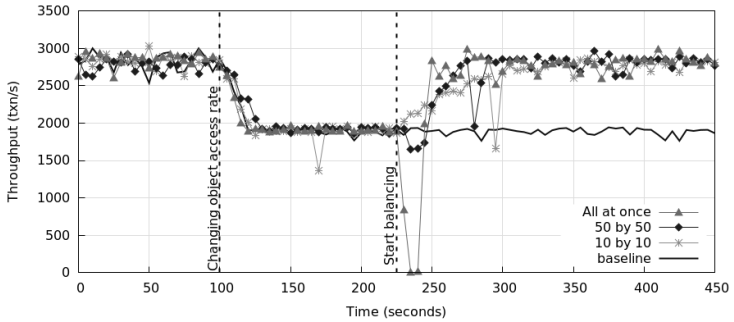


Figure 5: Throughput of one datacenter during the balancing process of top-k objects.

ing 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. Note that part of the observed latency in the reconfiguration is an artifact of our concrete implementation. The migration of data between the two independent instances of SwiftCloud could be avoided if we have opted to re-implement the system from scratch. However, the reconfiguration costs cannot be completely avoided unless the CRDT implementation supports the concurrent use of state- and operation-based updates for the same object. If fact, if only one of the implementations can be active at a given time (for a given object), replicas still need to coordinate to implement the switching.

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 paper, 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 [6], detect statistically relevant shifts of system's metrics [13], or predict future workload trends [7].

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. Figure 6 shows the results of the experiment. As one can see, 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. One could better tune this parameter

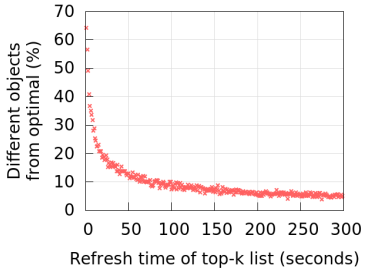


Figure 6: Percentage of objects in a top-k list that are different from the optimal top-k list.

if patterns of the workload are known beforehand.

6. 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 a 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 of 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 [1], 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 [10]) 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 [3].

• To use metadata maintenance techniques that avoid creating 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.

7. CONCLUSIONS

In this paper we have analyzed the cost performance tradeoffs between the two main approaches used to propagate updates in geo-replicated stores using CRDTs, namely operation- and state-based approaches. Our work shows that none of the approaches outperforms the other in absolute terms, and that a hybrid system may yield the best results.

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) the dissemination process for a bounded number of objects (hot objects). 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 it is capable of rapidly self-adapt to variations in the workload.

In this process, we have also been able to get interesting insights that may help in driving future improvements over BENDY. 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.

8. REFERENCES

[1] P. Almeida, A. Shoker, and C. Baquero. Efficient state-based CRDTs by delta-mutation. *CoRR*, abs/1410.2803, 2014.

[2] C. Baquero, P. Almeida, and A. Shoker. Making operation-based CRDTs operation-based. In K. Magoutis and P. Pietzuch, editors, *Distributed Applications and Interoperable Systems*, LNCS, pages 126–140. Springer Berlin Heidelberg, 2014.

[3] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.

[4] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[5] 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 *Middleware '15*, Vancouver, BC, Canada, 2015.

[6] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004.

[7] R. Kalman. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering*, 82(1):35–45, 1960.

[8] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[9] M. Letia, N. Preguiça, and M. Shapiro. CRDTs: Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009.

[10] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[11] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, volume 3363 of *LNCS*, pages 398–412. Springer, 2005.

[12] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro. Incremental stream processing using computational conflict-free replicated data types. CloudDP, 2013.

[13] E. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):pp. 100–115, 1954.

[14] N. Preguiça, J. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, 2009.

[15] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, 2011.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. SSS, 2011.

[17] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[18] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. SOSP, 2013.

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

[20] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. Middleware, 2015.