Observable Atomic Consistency for CvRDTs

Xin Zhao
KTH Royal Institute of Technology
Stockholm, Sweden
xizhao@kth.se

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Abstract

The development of distributed systems requires developers to balance the need for consistency, availability, and partition tolerance. Conflict-free replicated data types (CRDTs) are widely used in eventually consistent systems to reduce concurrency control. However, CRDTs lack consistent totally-ordered operations which can make them difficult to use.

In this paper, we propose a new consistency protocol, the observable atomic consistency protocol (OACP). OACP enables a principled relaxation of strong consistency to improve performance in specific scenarios. OACP combines the advantages of mergeable data types, specifically, convergent replicated data types, and reliable total order broadcast to provide on-demand strong consistency. By providing observable atomic consistency, OACP avoids the anomalies of related protocols.

We provide a distributed implementation of OACP based on Akka, a widely-used actor-based middleware. Our experimental evaluation shows that OACP can reduce coordination overhead compared to other protocols providing atomic consistency. Our results also suggest that OACP increases availability through mergeable data types and provides acceptable latency for achieving strong consistency.

CCS Concepts • **Computing methodologies → Distributed programming languages**;

Keywords atomic consistency, eventual consistency, actor model, distributed programming

ACM Reference Format:
Xin Zhao and Philipp Haller. 2018. Observable Atomic Consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE '18), November 5, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281366.3281372

AGERE '18, November 5, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6066-1/18/11…$15.00
https://doi.org/10.1145/3281366.3281372

1 Introduction

Conflict-free replicated data types (CRDTs) [35, 36] are widely used in industrial distributed systems such as Riak [5, 9] and Cassandra. They are objects which can be updated concurrently without consensus and converge to the same state if all updates are executed by all replicas eventually. Thus, they can provide high availability and scalability for replicated shared data.

There are two principal categories of CRDTs: operation-based CRDTs and state-based CRDTs. In operation-based CRDTs, also called CmRDTs, replicas propagate commutative update operations to other replicas, in order to guarantee eventual convergence. In contrast, in state-based CRDTs, also called CvRDTs, replicas propagate the entire state of the CRDT to other replicas whenever the state is updated; commutative functions are used to merge multiple revisions of the state of a CRDT.

However, the main challenge of programming with CRDTs is the fact that they only provide eventual consistency and they necessarily provide only a restricted set of operations. In particular, CRDTs do not support consistent non-convergent operations; for example, read operations may return outdated values before the replicas have converged, which makes the usage of CRDTs difficult.

In this paper, we address this challenge by extending CvRDTs with on-demand strong consistency. A novel protocol, the observable atomic consistency protocol (OACP), is used to guarantee *observable* atomic consistency for CvRDTs extended with totally-ordered operations. Totally-ordered operations provide on-demand atomic consistency, enabling consistent reads and consistent updates which do not have to be convergent. As a result, the extended CvRDTs lift an important restriction of CvRDTs, simplifying their usage.

1.1 Contributions

This paper makes the following contributions:

1. We introduce the observable atomic consistency (OAC) model which enables a novel extension of CvRDTs with consistent non-convergent operations. We lift a major limitation of CvRDTs, and significantly simplify programming with CvRDTs. We also provide a precise and formal definition of the OAC model.
2. We prove that systems providing OAC are state convergent. The paper summarizes our definitions and results, while our companion technical report [41] contains the complete proofs.

3. We introduce the observable atomic consistency protocol (OACP) which guarantees observable atomic consistency. We provide a distributed, cluster-enabled implementation of OACP on GitHub [40].
4. We experimentally evaluate OACP including latency, throughput, and coordination. Our evaluation shows how much OACP benefits from commutative operations, reducing the number of exchanged protocol messages compared to baseline protocols. We also experimentally evaluate an optimization of OACP.

The rest of the paper is organized as follows. Section 2 illustrates how application programmers use OACP. In Section 3 we formalize the observable atomic consistency (OAC) model, and prove that systems providing OAC are state-convergent. Section 4 explains the observable atomic consistency protocol (OACP). In Section 5 we present a performance evaluation of an actor-based implementation of OACP using microbenchmarks as well as a Twitter-like application. Section 6 discusses related work, and Section 7 concludes.

2 Overview

We provide an overview of the OACP system from the perspective of a programmer to provide a user-friendly interface towards distributed application development. First, we introduce the system structure, and then we demonstrate how to use the provided API through an example.

System structure. Figure 1 shows the system structure. There are three layers from bottom to top: storage, distributed protocol, and application. We first focus on the top layer and discuss how applications interface with the protocol.

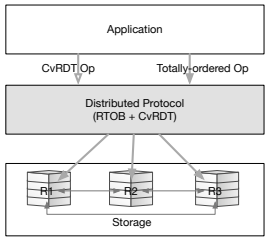


Figure 1. High-level view of OACP

The operations submitted by the application are divided into two categories: CvRDT operations (CvOps) and totally-ordered operations (TOps). CvOps are commutative. TOps are supported by reliable total order broadcast (RTOB) [13], so that their ordering is preserved across the entire system.

The submission of a TOp causes all replicas to atomically (a) synchronize their convergent states, and (b) lock their convergent states. A replica with locked convergent state buffers CvOps until the original TOp has been committed. Moreover, at the point when a TOp is executed, all replicas are guaranteed to have consistent convergent states. Thus, submitting a TOp ensures the consistency of all replicas, including their convergent states. We give a resettable counter as an example.

Resettable counter. The grow-only counter (GCounter) is one of the most basic counters which is widely used, e.g., in real-time analytics or in distributed gaming. It is a CvRDT which only supports increment and merge operations. However, when a system employs a GCounter to achieve eventual consistency, often a special "reset" operation is needed for resetting the counter to its default initial state. In other words, we need an "observed-reset" [4] operation for GCounter to go back to the bottom state, which means when "reset" is invoked, all effects of the GCounter observed in different replicas should be equivalently reset. However, the standard GCounter cannot solve this problem; this limitation is also well-known in the popular Riak DT implementation.

Thus, we need an implementation of a resettable counter to make sure that all the replicas are reset at the same time. A straightforward solution is to define "reset" as a totally-ordered operation, leveraging the property of TOps.

Figure 2 shows a GCounter definition in Scala. We extend the `CvRDT` trait to have an instance of GCounter which supports operations such as `incr` and `merge`. Each replica in the cluster is assigned an ID; this enables each `GCounter` instance to increment locally. When merging the states of two GCounters, we take the maximum counter of each index. The `compare` method is used to express the partial order relationship between different `GCounters`.

```
1  trait CvRDT[T] {
2    def myID(): Int
3    def merge(other: T): Unit
4    def compare(other: T): Boolean
5  }
6
7  abstract class GCounter extends CvRDT[GCounter] {
8    val p: Array[Int] = Array.ofDim[Int](3)
9    def incr(): Unit = {
10     val id = myID()
11     p(id) = p(id) + 1
12   }
13   def merge(other: GCounter): Unit =
14     for (i <- 0 until p.length)
15       p(i) = math.max(p(i), other.p(i))
16   def compare(other: GCounter): Boolean =
17     (0 until p.length).forall(i => p(i) <= other.p(i))
18 }
```

Figure 2. GCounter in Scala.

Actor-based user API. In the following, we illustrate the user API based on actors. The user-facing API consists of the following three parts.

- trait `CvRDT[T]`

- CvOp (CvValue, CvUpdate)
- TOp (TUpdate)

The `CvRDT` trait (A trait in Scala can be thought of as similar to a Java interface enabling a safe form of multiple inheritance) exposes the implementation of CvRDT to developers which allows them to define their own CvRDT types and operations which are provided by the trait such as initialisation, add, remove and merge. The last two methods provide access to use the OACP, allowing developers to decide whether the operation should be operated as CvRDTs or ordered messages. CvOps are used when one wants to gain benefits from high availability since they are directly sent to the closest available server. TOps are good options if the developer wants to make all the replicas reach the same state for some essential operations. These two message types give developers more flexibility when they want to achieve certain consistency as well as the performance of the implementation.

The objects are expressed using actors since actors are suitable for expressing concurrent computation using messages. The message handlers provided by the system need to be defined on the client side so that the OACP system can recognize the corresponding operation type. We extend the client actor with the OACP protocol to connect it to the application layer. The developer only needs to define which message to send using the following "Akka-style" message handler:

```
1  class CounterClient extends Protocol[GCounter] {
2    val CounterClientBehavior: Receive = {
3      case Incr  => self forward CvOp("incr")
4      case Get   => self forward TOp("Get")
5      case Reset => self forward TOp("Reset")
6      ...
7    }
8    override def receive =
9      CounterClientBehavior.orElse(super.receive)
10   ...
11 }
```

When the `CounterClient` receives a certain message, it behaves according to the user's definition. For example, when the received message matches case `Incr`, the client actor forwards a CvOp message to the protocol layer. In this way, the concrete implementation of the system is hidden from the developers.

Server-side stores a log which contains CvRDT states and the sequence of totally-ordered operations. The registry does not record submitted CvOps, and only when a TOp is executed the current CvRDT state will be stored together with the TOp label in the log entry. The log might look like the following:

When the user requires log from server-side, the counter value is the difference between the latest CvRDT state and the most recent CvRDT state with `Reset` label.

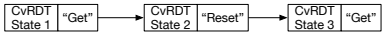


Figure 3. Log abstraction in OACP.

3 Observable Atomic Consistency

Definition 3.1 (CvT order)**.** Given a set of operations $U = C \cup T$ where $C \cap T = \varnothing$, a CvT order is a partial order $O = (U, \prec)$ with the following restrictions:

- $\forall u, v \in T$ such that $u \neq v.\ u \prec v \vee v \prec u$
- $\forall p \in C, u \in T.\ p \prec u \vee u \prec p$

According to the transitivity of the partial order, we could derive that $\forall l, m, n \in U$ such that $l \prec m, m \prec n.\ l \prec n$.

Definition 3.2 (Cv-set)**.** Given a set of operations $U = C \cup T$ where $C \cap T = \varnothing$, a Cv-set C_i is a set of C operations with the restriction that:

- $\forall p, q \in C_i \Rightarrow p \not\prec q \wedge q \not\prec p$
- $\forall p \in C \setminus C_i. \exists q \in C_i$ such that $p \prec q \vee q \prec p$

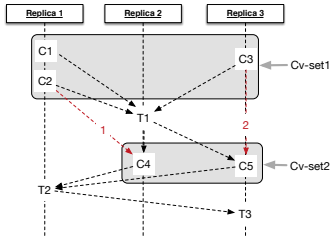


Figure 4. CvT order of operations.

In Figure 4, the partial order is labeled with black dash arrow, while we could derive line 1 and 2 (red dash arrow) from the transitivity of $\prec$. There are two different Cv-sets in this situation while the CvRDT updates inside have no partial order relationship.

In CvT order, only the operations in one same Cv-set could happen concurrently. Now we consider the operations on different sites. Suppose each site i executes a linear extension compatible with the CvT order. Then, the replicated system with n sites provides local atomic consistency, which is defined as follows.

Definition 3.3 (Local atomic consistency (LAC))**.** A replicated system provides local atomic consistency (LAC) if each site i applies operations according to a linear extension of the CvT order.

The three replicas in Figure 4 could have different linear extensions of the CvT order. However, despite possible reorderings, LAC guarantees state convergence.

Definition 3.4 (State convergence)**.** A LAC system is state convergent if all linear extensions of the underlying CvT order O reach the same state S.

Theorem 3.5. *Given a CvT order, if all operations in each Cv-set are commutative, then any LAC system is state convergent.*

A complete proof of the theorem is included in our companion technical report [41].

Definition 3.6 (Observable atomic consistency)**.** A replicated system provides observable atomic consistency (OAC) if it provides local atomic consistency and for all $p \in C$, $u \in T$. (a) $p \prec_{PO} u$ in program order (of some client) implies that $p \prec u$ in the CvT order, and (b) $u \prec_{PO} p$ in program order (of some client) implies that $u \prec p$ in the CvT order.

The constraints for OAC guarantee that the system state is consistent with the order of operations on each client side. Since OAC is a special case for LAC, the state convergence also holds for OAC, and we have the following corollary:

Corollary 3.7. *Given a CvT order, if all operations in each Cv-set are commutative, then any OAC system is state convergent.*

Comparison to RedBlue Consistency. A closely related consistency model is RedBlue consistency [27]. Red operations are the ones which need to be totally ordered while the blue ones can commute globally. Two operations can commute means that the order of the operations does not affect the final result. In order to adapt an existing system to a RedBlue system, *shadow operations* with commutativity property need to be created and then adjust the application to use these shadow operations. The shadow operations can "introduce some surprising anomalies to a user experience" [27]. In other words, the final system state might not take all messages into account that were received by the different replicas. Thus, RedBlue consistency can only maintain a local view of the system. In OAC, the CvRDT updates can only commute inside a specific scope, namely, between two totally-ordered operations. This restricts the flexibility of CvRDT updates, but at the same time, it provides a consistent view of the state. Importantly, the final system state always matches the state resulting from a linear extension of the original partial order of the operations. An example comparing OAC and RedBlue consistency is included in our companion technical report [41].

4 Observable Atomic Consistency Protocol

Following the definition of observable atomic consistency (See Definition 3.6), we now introduce a protocol that enforces this notion of consistency. We present the observable atomic consistency protocol (OACP) in four steps: first, we describe the so-called *data model*, similar to prior related work [12]; second, we describe the client-side protocol; third, we describe the server-side protocol; finally, we discuss differences to the most closely related previous protocol.

Data model. The function $Eval : Op^* \times Op \rightarrow Value$ obtains the result of applying a sequence of operations to a single object. *Value* is an abstract data type representing the result of reading the log. We define three different cases for the *Op* abstract data type: *CvUpdate* represents CvRDT updates, *TUpdate* represents totally-ordered updates, and *Read* represents totally-ordered read operations. The data model is then given by the function $OACPVal : Read \times \{CvUpdate, TUpdate\}^* \rightarrow Value$.

In the following, we use the type *Log* to represent a log that records the totally-ordered sequence of all operations and states. Whenever a totally-ordered operation is committed, an entry is created and added to the log. Each log entry is of type `Entry` which is defined as follows:

```
class Entry {
  origin: Client, nextLogIndex: N, cState: CvRDT, op: TOp }
```

Each entry contains a reference to the client that submitted the TOp (`origin`); the index of the next log entry (`nextLogIndex`); the state of the underlying CvRDT (`cState`); and the committed totally-ordered operation (`op`). We use the notation ++ to represent the concatenation of two sequences.

OACP client protocol. We now show the basic version of the OACP client-side protocol in Listing 1. When invoking a CvOp, the client sends the update to a random available server. When invoking a TOp, the client submits the update to the current leader according to the chosen consensus protocol (in our case, Raft [34]).

Listing 1. Client-side OACP pseudocode.

```
1  role OACP_Client {
2    var result: Promise[Value];
3    var response: Promise[Value];
4
5    // client interface
6    CvOp(u: CvUpdate) {
7      response := new Promise[Value];
8      CRDT_submit(u); // send to random replica
9      return response;
10   }
11
12   TOp(msg: TUpdate | Read): Promise[Value] {
13     result := new Promise[Value];
14     RTOB_submit(msg); // send to leader replica
15     return result;
16   }
17
18   // network interface
19   onReceive(log: Log) {
20     result.complete(OACPVal(updates(log)));
21   }
22
23   onReceive(response) {
24     response.complete()
25   }
26
27   function updates(l: Log): (TUpdate*, cState) =
```

```
28      return (l[0].tUpdate ++ ... ++ l[l.length-1].tUpdate,
29              l.cState);
30  }
```

An important assumption of the protocol is that the same client never invokes an operation before the promise of a previous invocation has been completed. In particular, when a TOp happens directly after a CvOp, the client awaits an acknowledgment of the previous message from the server side before invoking the TOp. This means the program order on the client side is preserved on the server side.

OACP server protocol. We now move on to the server side of the protocol. We assume that the application has continuous access to the network since this mirrors the practical usage of many existing applications, such as chat and Twitter-like micro-blogging.

In Listing 2, *CRDT.merge* (line 11) is a merge operation for the abstract *CRDT* type; *Log.nextIndex* (line 41) is the next store index for *Entry* according to the Raft consensus protocol; *Broadcast*() (line 12) is a gossip message which is sent to all other replicas; the *RTOB* function consistently appends the argument *Entry* to the logs of all replicas.

When the server receives a CvOp, it merges the current CvRDT state and broadcasts the change to the other servers. When a TOp is received, the leader server collects the current states from the other replicas and performs an RTOB, so that each replica maintains the same, consistent log. Compared with CvOp, TOp requires one RTOB together with $2(n-1)$ gossip messages (where n is the number of replicas).

In our OACP reference implementation, RTOB is implemented by the Raft distributed consensus protocol. When *TUpdate* and *Read* are handled by a non-leader server, the server forwards the update to the current leader. Moreover, we define *Read* as a TOp to guarantee strong consistency.

The protocol description contains a special flag "frozen". This flag is set to true during the processing of a TOp, on each replica (lines 20 and 30). This means that subsequent operations are stashed (lines 9 and 18), and the current state remains unchanged. Only when the TOp has been committed is the flag reset and previously stashed messages are put back into the message queue.

Listing 2. Server-side OACP pseudocode (unoptimized).

```
1  role OACP_Server {
2    var currentState: CRDT;
3    var log: Log;
4    var currentLeader: Server;
5    var frozen: Boolean;
6    var result: Promise[Value];
7
8    onReceive(u: CvUpdate) {
9      if frozen then { buffer.stash(u); }
10     else {
11       currentState = CRDT.merge(currentState, u);
12       Broadcast(currentState);
13       client.reply(); //acknowledge to client
14     }
15   }
16
17   onReceive(msg: TUpdate | Read) {
18     if currentRol.isLeader && frozen then { stash(msg); }
19     else if (currentRole.isLeader) {
20       frozen = true;
21       numStateMsgReceived = 0;
22       result = new Promise[Value];
23       result.onSuccess { v => client.reply(v);}
24       Broadcast(GetState);
25     }
26     else {forward(currentLeader, msg);}
27   }
28
29   onReceive(msg: GetState) {
30     frozen = true;
31     reply(StateIs(currentState));
32   }
33
34   onReceive(msg: StateIs) {
35     if currentRole.isLeader then {
36       numStateMsgReceived += 1;
37       currentState = CRDT.merge(currentState, msg.cState);
38       if numStateMsgReceived == numReplicas-1 then {
39         RTOB(new Entry {
40             origin = msg.sender,
41             number = Log.nextIndex(log),
42             cState = currentState,
43             toUpdate = msg });
44         Broadcast(Melt);
45         result.complete(log);
46       }
47       if timeout then { // fault handler
48         RTOB(new Entry {
49             origin = msg.sender,
50             number = Log.nextIndex(log),
51             cState = Log.cState,
52             toUpdate = Recovery });
53         Broadcast(Melt);
54         result.complete(failure);
55         }
56       }
57     }
58   }
59
60   onReceive(msg: Melt) {
61     frozen = false;
62     buffer.unstash() or discard();
63   }
64 }
```

When the leader receives acknowledgments from a majority of servers, a "Melt" message (line 60) is broadcast to reset the "frozen" flag to false. If there are n servers in the cluster, then there are $2(n-1)$ messages added to the messages exchanged by the protocol. Therefore, we devised an optimized

version (see Listing 3) to reset the flag to false each time a server makes a commit in the consensus protocol.

Listing 3. Server-side OACP pseudocode (optimized)

```
1  onCommit(e: Entry) {
2    frozen = false;
3    buffer.unstash() or discard();
4  }
```

Fault Model. During the processing of an RTOB, crash failures of n nodes are tolerated in a cluster with $2n + 1$ nodes using Raft. During the state gathering phase (line 38), if the receiving time of the leader exceeds a timeout (line 47), then the failure recovery strategy is to retrieve the `cState` in the last log entry and to synchronize all replicas. This strategy will cause observable data loss for the CvOps that are sent to failed replica, but it avoids inconsistent data.

Actor-based Implementation. Different mechanisms can be used to implement the protocol based on the abstraction above. In particular, the protocol can be directly mapped to an actor-based implementation, since asynchronous messages can be used for the submission of operations (CvOps and TOps), and the `onReceive` protocol methods can be expressed using actor message handlers.

Comparison to GSP. GSP [12, 33] is an operational model for replicated shared data. It supports update and read operations from the client. Update operations are stored both in the local pending buffer so that when a read happens, it will perform "read your own writes" directly from the local storage. That property makes offline read possible so that even when the network is broken, the application can work properly and the following updates will be stashed in the buffer and resent until the network is recovered.

GSP also relies on RTOB to send back a totally ordered sequence of updates to the locally known buffer. The existence of known and pending buffers provides the possibility for updating these buffers fully asynchronously. In order to achieve high throughput, it also provides the batching option so that it does not require RTOB for every operation. To support the offline property, GSP performs local reads.

However, the local read operations provide some confusing conditions such as the following from paper [12]:

$wr(A, 2)$	.
.	$wr(B, 1)$
.	$wr(A, 1)$
.	$rd(A) \rightarrow 2$
$rd(B) \rightarrow 0$	.

Above is the key-value store shared data model which initially stores 0 for each address. In GSP protocol, such interleaving will be possible since $rd(B)$ can get 0 before the local storage gets the update from $wr(B, 1)$.

	Ohio	London	Sydney
Ohio	0.53ms	85.6ms	194ms
London		0.42ms	279ms
Sydney			0.88ms

Figure 5. Average round-trip latency between Amazon sites

In OACP, we process read as a TOp and it always gets the updated state from the server. Thus, only the following is possible, and the observable strong consistency is preserved:

$wr(A, 2)$	.
.	$wr(B, 1)$
.	$wr(A, 1)$
.	$rd(A) \rightarrow 1$
$rd(B) \rightarrow 1$	.

In GSP, one can also execute all reads as linearizable reads which is equivalent to OACP reads. However, it requires additional implementation for read operations, and all the operations in GSP will need RTOB which has a high cost.

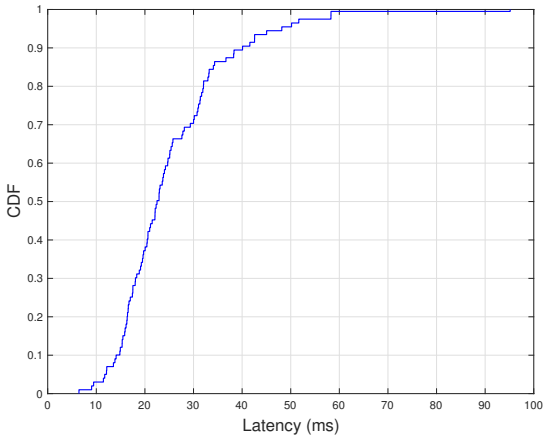
5 Performance Evaluation

We evaluate the performance of OACP from the perspective of latency, throughput, and coordination. We use a microbenchmark as well as a Twitter-like application inspired by Twissandra [39].

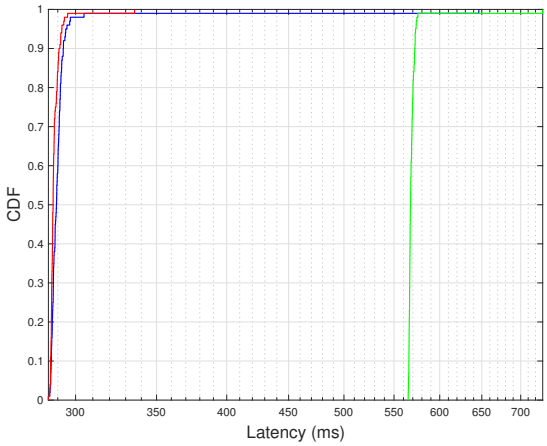
Experimental set-up. Our OACP implementation is based on the cluster extension of the widely-used Akka [29]. The cluster environment is configured using three seed nodes; each seed node runs an actor that detects changes in cluster membership (i.e., nodes joining or leaving the cluster). This enables freely adding and removing cluster nodes.

The experiments on coordination are performed on a 1.6 GHz Intel Core i5 processor with 8 GB 1600 MHz DDR3 memory running macOS 10.13. The experiments on latency and throughput are performed on Amazon EC2 cluster which includes three T2 micro instances (1 vCPU, 1 Gb memory, and EBS-only storage.) running in Ohio, London, and Syndey. Figure 5 shows the average round-trip latency between each pair of sites. Our implementation, available open-source [40], is based on Scala 2.11.8, Akka 2.4.12, AspectJ 1.8.10, and JDK 1.8.0_162 (build 25.162-b01).

Latency. In OACP, any CvOp gets an immediate response once the request arrives at any of the servers in the cluster. In contrast, the TOp runs consensus protocol underneath to achieve consistency. In order to understand the effect of CvOps and TOps, we measure the latency for CvOps and TOps on Amazon EC2 in three different regions: Ohio, London and Sydney and we plot the CDFs of observed latencies in Figure 6. The cluster in our experiments consists of three nodes which locate in different regions. A leader node needs to be elected to keep consensus. We put the client node in London and measure different conditions when the leader



(a) CvOp latency



(b) TOp latency

Figure 6. Latency CDF for CvOps and TOps when leader node locates on different regions. In (b), the green (the right most) line corresponds to the condition where leader locates in Sydney, the blue line corresponds to Ohio and the red (the left most) line corresponds to London.

node locates in different regions. In general, CvOps get quick responses as we can see that the maximum latency is 60 ms, and 90% of the response latency is within 40 ms. The latency of TOps depends on the location of the leader node. When the leader node locates in Sydney, the maximum latency is 600 ms. Moreover, when the leader node locates in the other two regions, the maximum latency is around 350ms.

Throughput. Now we focus on the throughput of OACP. We generate benchmarks with different proportion of CvOps and TOps. While we increase the number of concurrent requests to the same consistent log in the cluster, we measure the duration for processing all of the requests. The throughput is then the number of requests divided by the duration. The results in Figure 7 show that increasing the ratio of TOps decreases the throughput. TOps require the leader node to force all the other replicas to reach consensus on the same

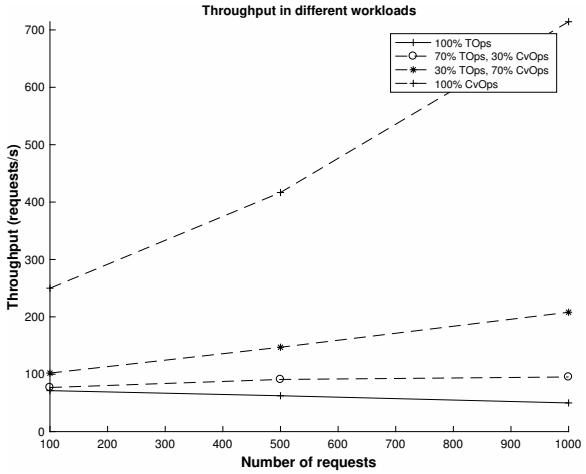


Figure 7. Throughput for a 3 site cluster with varying CvOp and TOp workload mixes.

log. Thus there will be a request queue on the server side. Any of the server nodes can process CvOps, and the commutative property of CvOps allows them to be processed in a random order. The throughput of the mix workloads is located between the pure workloads which give the programmer a range of choices.

Coordination. We consider this aspect because previous work has shown that reducing the coordination within the protocol can improve the throughput of user operations dramatically [7]. In order to evaluate the performance independent of specific hardware and cluster configurations, our experiments count *the number of exchanged messages*. The message counting logic is added via automatic instrumentation of the executed JVM bytecode using AspectJ [21].

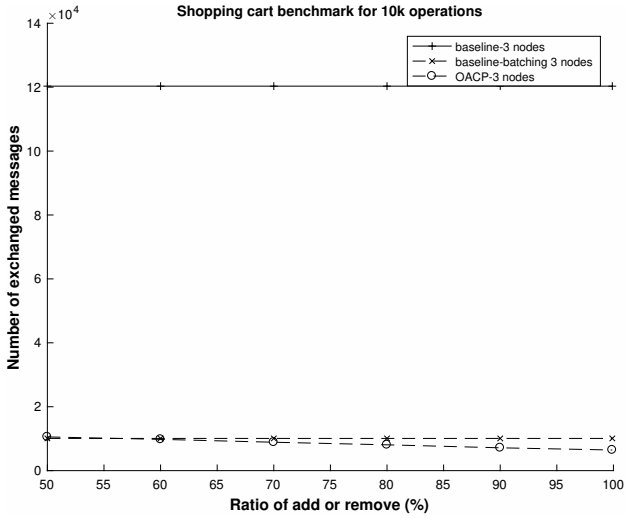
5.1 Microbenchmark

We start the evaluation with a simple shopping cart benchmark to see the advantages and weakness of the OACP protocol. We define the "add" and the "remove" operation as CvOps in OACP. The "checkout" operation in OACP is defined as a TOp (to ensure consistency upon checkout). Then we generate sequences of n operations where $n \in (0, 1000]$; each operation can be either "add", "remove", or "checkout".

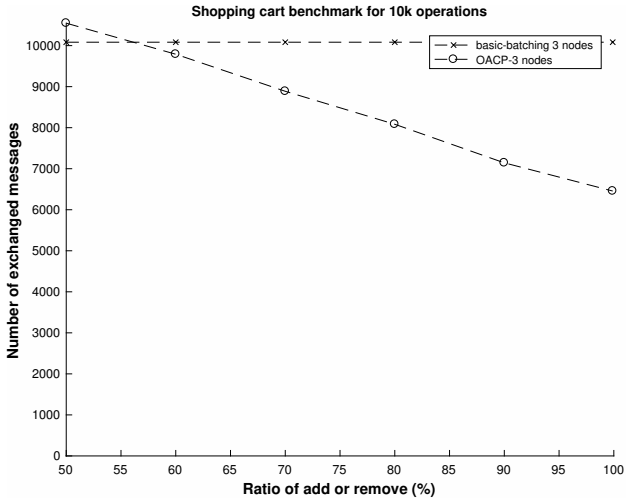
We compare the performance of OACP with the other two baseline protocols which are described as follows.

- Baseline protocol: every operation is submitted using RTOB to keep consistency on all the replicas.
- Baseline protocol with batching: an optimized version which gives a fixed batching buffer and allows to submit multiple buffered operations at once.
- OACP protocol: the protocol as described in Section 4.

We compared the number of exchanged messages among the baseline protocol, the batching protocol (batching buffer



(a)



(b)

Figure 8. Comparison on coordination among different protocols

size: 5000 operations), and the OACP protocol, using a 3-node cluster. The results are shown in Figure 8a.

The x-axis represents the ratio of "add" or "remove" operations in the whole sequence of operations (10k requests in this case), the y-axis represents the number of exchanged messages. From Figure 8a, we can see that the baseline protocol requires a much higher number of exchanged messages, namely 12× the number of messages compared to both the batching protocol and the OACP protocol. In the more detailed Figure 8b, we can see that when the ratio of CvOps increases, OACP benefits more. In OACP, when CvOp is 90%, the number of messages can be reduced by 30% compared with the batching protocol.

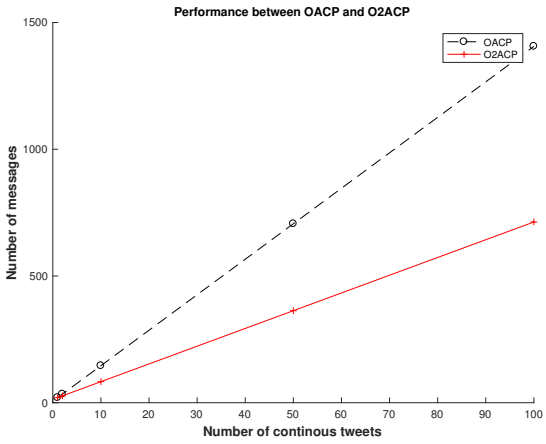
These results suggest the following *guidelines* for helping developers choose which protocol to use. When the percentage of CvOps is low (less than 50% in the above microbenchmark), i.e., when the application needs TOps quite frequently, then batching for TOps is a better choice. When more CvOps are happening between two TOps, then OACP performs better.

5.2 Case Study: Twitter-Like Application

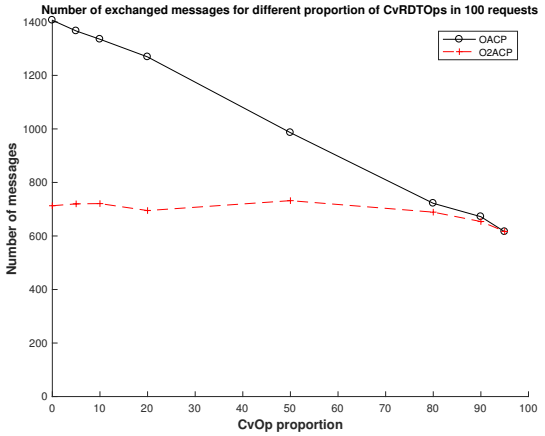
Following the shopping cart microbenchmark, we now extend our experiments to a more realistic application. This also allows us to investigate more aspects of our system since the application makes use of all features of OACP. We define a simple Twitter-like social networking application which supports AddFollower, Tweet, and Read operations. We define Tweet and Read as TOps and AddFollower as a CvOp in OACP. In the benchmarks, we focus on one specific user with a certain number of followers and send tweets.

Optimized observable atomic consistency protocol. For Twitter-like applications, the frequency of different events varies for different users. Some popular accounts tend to tweet more and to follow fewer users, while some newcomers follow more and tweet less. Consider the following operation sequence:
(1) Tweet → (2) Tweet → (3) AddFollower → (4) Tweet.
Since we define Tweet as a TOp, the state of all replicas is consistent after each Tweet operation. Therefore, there is no need for a merge operation to be executed between (1) and (2). However, in the case (3) → (4), the state updated by (3) first needs to be merged into all replicas before executing (4). Thus, one possible optimization is to notify the system of the sequence of operations, so that when two TOps happen consecutively (e.g., (1) → (2)), the OACP system does not need to gather state information first. In this way, the system can decrease the number of exchanged messages. We call this optimization O^{2}ACP.

In order to make performance comparison to the original OACP, we generate random sequences of operations according to the proportion of CvOps and then excute through both protocols. The results are shown in Figure 9a. When the proportion of tweets grows, the increase of messages is significantly smaller in the case of O^{2}ACP than in the case of OACP. In the case of 100 tweets, O^{2}ACP only requires about 50% of the messages of OACP, which is a significant improvement. We made another measurement for random sequences of TOps and CvOps, by varying the proportion of CvOps between 0% and 95%. The total number of client requests is 100. The results are shown in Figure 9b. In each case, we take the average of 10 measurements for each proportion. The proportion of CvOps has a more substantial effect in the case of OACP than in the case of O^{2}ACP, since when CvOps increase from 0% to 95%, the exchanged messages in



(a)



(b)

Figure 9. OACP vs. O^2ACP

OACP reduce from around 1400 to 600, while in O^2ACP, the number of exchanged messages remains quite stable.

6 Related Work

Consistency levels. The CAP theorem [14] points out the impossibility for any distributed system to achieve consistency, availability and partition tolerance at the same time. Zookeeper [18] provides sequential consistency [15] such that updates from a client are applied in the order in which they were sent. Bayou [38] is designed for supporting real-time collaborative applications and thus gives up consistency for high availability, providing eventual consistency [11]. Consistency Rationing [22] allows a user to define the consistency guarantees as well as switch the guarantees at runtime automatically. There are also many classifications in multi-level consistency. Fork consistency [28, 32] allows users to read from *forked sites* that may not be up-to-date. In contrast, write operations require all sites to be updated. Lazy replication [23] provides a solution to ensure high availability while keeping the data consistent. It divides updates into three categories: causal, forced, and immediate. The immediate operation is equivalent to our totally-ordered operation. RedBlue consistency [27] is closely related to observable atomic consistency; we provide a comparison at the end of Section 3. SIEVE [26] is a system based on RedBlue consistency which automatically chooses consistency levels according to a user's definition of system invariants. Explicit consistency in Indigo [3] guarantees the preservation of specific invariants to strengthen consistency beyond eventual consistency.

Distributed application development frameworks. Correctables [16] is an abstraction to decouple applications from their underlying database, which also provides incremental consistency guarantees to compose multiple consistency levels. QUELEA [37] enables programmers to provide fine-grained application-level consistency constraints which are mapped automatically to the guarantees ensured by the underlying data store. GSP [12, 33] provides an operational reference model for replicated shared data. It abstracts from the data model so that it can be applied to different kinds of data structures. A comparison between GSP and OACP is mentioned in Section 4. Orleans [6] provides a *virtual actor* abstraction for modeling and implementing distributed systems. The Akka framework [17, 29], which we use to implement OACP (see Section 5) provides a widely-used implementation of the actor model [1] on the JVM for writing highly concurrent, distributed, and resilient applications.

Data management in distributed systems. Paxos [25] and Raft [34] are popular consensus protocols in replicated systems and serve as the basis for reliable total-order broadcast [13]. The ZooKeeper Atomic Broadcast protocol [19] (Zab) guarantees the replication order in ZooKeeper using Paxos. Our RTOB implementation is inspired by Zab but uses Raft because of its simplicity. For resolving shared-data conflicts in distributed systems, there are several approaches. CRDTs [35] and cloud types [8, 10] resolve conflicts automatically using convergent operations, but they impose important restrictions on data structures. Cassandra [24], CaCOPS [30], Eiger [31], and ChainReaction [2] use the last-write-wins strategy to ensure availability; however, they may lose data if concurrent writes happen frequently enough. Riak [9] and mergeable types [20] provide the ability to resolve write conflicts on the application level.

7 Conclusion

We introduced the observable atomic consistency (OAC) model which enables a new extension of CvRDTs with totally-ordered operations. While lifting a main limitation of CvRDTs, we believe that it can significantly simplify programming with CvRDTs. We presented a proof of state convergence for systems providing observable atomic consistency. We then discussed a new consistency protocol, called observable

atomic consistency protocol (OACP), which guarantees OAC for distributed systems, and reduces the ambiguous consistency guarantee caused by local read in global sequence protocol. Experimental results show that OACP can reduce the coordination compared to the baseline consistency protocol in several microbenchmarks and there are space for optimization on individual cases.

Acknowledgments

This work is partially supported by Chinese Scholarship Council, the National Key Research and Development Program of China (2016YFB0200401), by program for New Century Excellent Talents in University, by the HUNAN Province Science Foundation 2017RS3045.

References

[1] Gul A. Agha. 1986. *ACTORS: A Model of Concurrent Computation in Distributed Systems.*

[2] Sérgio Almeida, João Leitão, and Luís E. T. Rodrigues. 2013. ChainReaction: a causal+ consistent datastore based on chain replication. In *EuroSys*. 85–98.

[3] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *EuroSys*. 6:1–6:16.

[4] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. 2016. The problem with embedded CRDT counters and a solution. In *EuroSys*. 10:1–10:3.

[5] Basho Technologies, Inc. 2012-2017. Riak DT source code repository. https://github.com/basho/riak_dt.

[6] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report.

[7] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-distribution of actor-based services. *PACMPL* 1, 107:1–107:26.

[8] Philip A. Bernstein and Sergey Bykov. 2016. Developing Cloud Services Using the Orleans Virtual Actor Model. *IEEE Internet Computing* 20, 5, 71–75.

[9] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. 2014. Riak DT map: a composable, convergent replicated dictionary. In *EuroSys*. 1:1.

[10] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *ECOOP*. 283–307.

[11] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. 2013. *Understanding eventual consistency.* Technical Report. Microsoft Research.

[12] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *ECOOP*. 568–590.

[13] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv* 36, 4 (2004), 372–421.

[14] Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* (2002), 51–59.

[15] J. R. Goodman. 1989. *Cache Consistency and Sequential Consistency.* Technical Report. IEEE Scalable Coherence Interface Working Group.

[16] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. *CoRR*.

[17] Philipp Haller. 2012. On the integration of the actor model in mainstream technologies: The Scala perspective. In *AGERE!@SPLASH*. 1–6.

[18] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*.

[19] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *DSN*. 245–256.

[20] Gowtham Kaki, KC Sivaramakrishnan, Samodya Abeysiriwardane, and Suresh Jagannathan. 2017. Mergeable Types. In *ML workshop*.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP*. 327–353.

[22] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay only when it matters. *PVLDB* 2, 1 (2009), 253–264.

[23] Rivka Ladin, Barbara Liskov, and Sanjay Ghemawat. 1992. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems* 10, 4 (Nov. 1992), 360–391.

[24] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 35–40.

[25] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst* 16, 2 (1998), 133–169.

[26] Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *USENIX Annual Technical Conference*. 281–292.

[27] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *OSDI*. 265–278.

[28] Jinyuan Li, Maxwell N. Krohn, David Mazieres, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *OSDI*. 121–136.

[29] Lightbend, Inc. 2009. Akka. http://akka.io/. Accessed: 2016-03-20.

[30] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*. 401–416.

[31] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI*.

[32] Mazieres and Shasha. 2002. Building Secure File Systems out of Byzantine Storage. In *PODC*.

[33] Hernán C. Melgratti and Christian Roldán. 2016. A Formal Analysis of the Global Sequence Protocol. In *COORDINATION*. 175–191.

[34] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*. 305–319.

[35] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Technical Report RR-7506; inria-00555588. HAL CCSD.

[36] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS*. 386–400.

[37] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *PLDI*. 413–424.

[38] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*. 172–183.

[39] Twissandra. 2014. Twitter clone on Cassandra. http://twissandra.com/. Accessed: 2018-02-26.

[40] Xin Zhao. 2018. OACP implementation source code repository. https://github.com/CynthiaZ92/OACP.

[41] Xin Zhao and Philipp Haller. 2018. *Observable atomic consistency for CvRDTs.* Technical Report. https://arxiv.org/abs/1802.09462