Journal of Logical and Algebraic Methods in Programming 114 (2020) 100561

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Replicated data types that unify eventual consistency and observable atomic consistency

Xin Zhao *, Philipp Haller

School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden

ARTICLE INFO

Article history:
Received 1 March 2019
Received in revised form 30 March 2020
Accepted 16 May 2020
Available online 22 May 2020

Keywords:
Atomic consistency
Eventual consistency
CRDT
Actor model
Distributed programming

ABSTRACT

Strong consistency is widely used in systems such as relational databases. In a distributed system, strong consistency ensures that all clients observe consistent data updates atomically on all servers. However, such systems need to sacrifice availability when synchronization occurs.

We propose a new consistency protocol, the observable atomic consistency protocol (OACP) to make write-dominant applications as fast as possible and as consistent as needed. OACP combines the advantages of (1) mergeable data types, specifically, convergent replicated data types, to reduce synchronization and (2) reliable total order broadcast to provide on-demand strong consistency. We also provide a high-level programming interface to improve the efficiency and correctness of distributed programming.

We present a formal, mechanized model of OACP in rewriting logic and verify key correctness properties using the model checking tool Maude. Furthermore, we provide a prototype implementation of OACP based on Akka, a widely-used actor-based middleware. Our experimental evaluation shows that OACP can reduce coordination overhead compared to the state-of-the-art Raft consensus protocol. Our results also suggest that OACP increases availability through mergeable data types and provides acceptable latency for achieving strong consistency, enabling a principled relaxation of strong consistency to improve performance.

1. Introduction

Strong consistency is needed for many applications. For example, a bank application should always be able to show the account balance to the user, which corresponds to the actual amount; an online bidding system can only have one winner at the end of each auction. However, strong consistency is expensive due to its high synchronization overhead. As a result, it might slow down the system or even block the system when the network is not available.

According to the CAP theorem [1], we can at best achieve eventual consistency in an available distributed system with fast responses and partition tolerance. Conflict-free replicated data types (CRDTs) [2,3] are widely used in industrial distributed systems such as Riak [4,5] and AntidoteDB [6]. They are suitable for implementing highly available and scalable

* Corresponding author.
E-mail addresses: xizhao@kth.se (X. Zhao), phaller@kth.se (P. Haller).

https://doi.org/10.1016/j.jlamp.2020.100561

replicated shared data since they support concurrent updates, and they are guaranteed to converge to the same state if all replicas eventually execute all updates.

In this work, we introduce a novel observable atomic consistency (OAC) model which provides both fast convergent updates and non-convergent operations requiring synchronization. The presented observable atomic consistency protocol (OACP) provides OAC, and can be implemented assuming only that the communication subsystem ensures eventual delivery; this assumption is shared by CRDTs, thus ensuring practicality. In summary, the integration of mergeable data types and strong synchronization enables fast convergent updates, consistent reads, and non-convergent operations, such as consistent updates.

1.1. Contributions

This paper makes the following contributions:

1. We provide a complete explanation of the observable atomic consistency (OAC) model, which supports fast convergent updates in a consistent setting. We also provide a precise and formal definition of the OAC model.
2. We prove that systems providing OAC are state-convergent. This result shows that under OAC, the states of all replicas are guaranteed to eventually be the same after executing the same set of operations.
3. We give a detailed description of the observable atomic consistency protocol (OACP), which guarantees observable atomic consistency. Our user-friendly distributed implementation of OACP based on Akka [7] is available on GitHub [8].
4. We provide a formal, mechanized model of OACP based on rewriting logic. Using the Maude [9] model checker, we mechanize three essential properties of OACP, namely (1) that the state of all replicas is made consistent upon executing a totally-ordered operation, (2) that the protocol preserves a specific execution order for the submitted operations, and (3) that the protocol does not generate ghost messages, i.e., messages that are never consumed. Using Maude, we have successfully applied model checking verification to explore the full, finite state space for several non-trivial test cases. Thus, model checking has significantly increased our confidence in the correctness of OACP.
5. We experimentally evaluate OACP, including latency, throughput, coordination, and scalability. Our evaluation shows how much OACP benefits from commutative operations, reducing the number of exchanged protocol messages compared to the state-of-the-art Raft consensus protocol. We also discuss an optimization of OACP and experimentally evaluate its performance improvements for a case study.

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). Section 5 contains a brief introduction to Maude, the key steps of modeling OACP in Maude, and the verification result for the model. In Section 6, we present the performance evaluation of an actor-based implementation of OACP using microbenchmarks as well as a Twitter-like application. Section 7 discusses related work, and Section 8 concludes and points out directions for future work.

An earlier, short version of this paper appeared at AGERE 2018 [10]. The present article has been significantly extended with (1) a formal proof of the state convergence of the protocol, (2) a formal model of the protocol based on rewriting logic mechanized using the Maude model checker (see Contribution 4), and (3) additional experiments evaluating the scalability of the system.

2. Overview

2.1. CvRDTs

There are two main categories of CRDTs: operation-based CRDTs and state-based CRDTs. Operation-based CRDTs propagate commutative update operations between replicas. They require “exactly once delivery”, which requires reliable causal broadcast. State-based CRDTs, also called CvRDTs, propagate the entire state of the CRDT between replicas whenever the state is updated. Commutative functions are used to merge multiple revisions of the state of a CRDT, and thus the state can be sent multiple times. Our work is based on state-based CRDTs (CvRDTs), because it is less expensive to provide reliable broadcast.

In order to illustrate our ideas in a clear and simple way, we start with a very basic CvRDT: the grow-only counter, also called GCounter.

Grow-only counter. The GCounter is one of the most basic counters which is widely used, e.g., in real-time analytics or distributed gaming. It only supports two operations: increment and merge.

Fig. 1 shows a GCounter definition in Scala [11]. Scala is an object-oriented, functional, and statically-typed language which targets the Java Virtual Machine (JVM) and JavaScript runtimes.[1] The keyword `trait` introduces an interface-like

[1] See https://www.scala-lang.org.

```
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 }
```

Fig. 1. GCounter in Scala.

class that can be mixed-in to other classes (via mixin composition [12]). Square brackets `[]` contain a type parameter section. The definitions within the trait consist of the three abstract methods `myID`, `merge`, and `compare`. A value declaration `val x: T` introduces *x* as a name of a value of type *T*; *x* cannot be re-assigned afterwards. Return type `Unit` corresponds to void methods in Java.

Each replica in the cluster is assigned an ID, and the `incr` operation enables each `GCounter` instance to increment the corresponding index locally. The `merge` operation merges the states of two `GCounter`s by taking the maximum counter of each index. The `compare` method is used to express the partial order relationship between different `GCounter`s.

When a system employs a GCounter to achieve eventual consistency, often, a particular "reset" operation is needed for resetting the counter to its default initial state. In other words, we need an "observed-reset" [13] 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, since each operation can only monotonically increase its internal state. This limitation is also well-known in the widespread Riak DT implementation [5].

2.2. Extended operations and user APIs

Besides CvRDTs, our framework includes some extended operations. We introduce these operations together with user APIs in the following section.

Extended operations. We refer to the operations that a CvRDT defines as CvOps (short for "convergent operations"). Since CvRDTs do not support non-convergent operations, we extend the system with totally-ordered operations and refer to them as TOps. TOps are supported by reliable total order broadcast (RTOB) [14], so that their ordering is preserved across all replicas.

CvOps are commutative according to the mathematical definition of CvRDTs, and can thus be concurrently processed by different replicas. In contrast, when a TOp is submitted, all replicas atomically (a) synchronize their convergent states and (b) lock their convergent states. Locking the convergent states means that during this phase of the protocol, CvOps are not processed but queued up for later execution. Atomicity is guaranteed using distributed consensus, which means that all replicas are guaranteed to have consistent convergent states when a TOp is executed. Thus, submitting a TOp ensures the consistency of all replicas, including their convergent states.

Using the extended operations, it is now possible for us to define a TOp called `Reset` to implement a resettable counter to enable resetting all replicas consistently.

Actor-based user API. In the following, we illustrate the user API based on actors [38]. The user-facing API consists of the following three parts:

- `trait CvRDT[T]`
- `CvOp(CvUpdate)`
- `TOp(TUpdate)`

The `CvRDT` trait[2] exposes the implementation of CvRDT to developers which allows them to define their own CvRDT types and operations such as initialization, add, remove and merge. The last two methods are provided for the developers to

[2] A trait in Scala can be thought of as similar to a Java interface enabling a limited form of multiple inheritances.

decide the operation's property. If one wants to gain benefits from high availability since they are directly sent to the closest available server, then more CvOps could be defined. If one wants to make all the replicas reach the same state, then a TOp should be used.

2.3. Resettable counter example

Step 1: Customize the definition of a CvRDT trait. In our framework, for example, one can define a GCounter as in Listing 1.

```
case object CounterCRDT extends CvRDT[Array[Int], Int, Int, VectorTime] {
  override def empty: Array[Int] = Array(0, 0, 0)

  override def apply(fun: (Array[Int], Int, Int, VectorTime) => Array[Int],
  crdt: Array[Int], a1: Int, a2: Int, a3: VectorTime): Array[Int] = {
    fun(crdt, a1, a2, a3)
  }

  override def merge(currentState: Array[Int], nextState: Array[Int]) = {
    ...
  }

  ...

  def incr = (data: Array[Int], id: Int, time: VectorTime) => {
    data(id) = data(id) + 1
    data
  }

}
```

Listing 1: Source code of GCounter definition.

The CRDT operation "incr" is defined as a Scala closure so that one can apply the user-defined function once they receive it.

Step 2: Define the extended totally-ordered operations. Based on the GCounter definition, for the resettable counter application, one can define extended totally-ordered operations as "Get" and "Reset". "Get" is a consistent read operation, and "Reset" is a consistent write operation. In our framework, the operations are sent to the server-side as commands stored on the server's log, so we use strings to represent them. We have a detailed explanation about how to process the logs in section 4.2.

Step 3: Implement client side handler. After the first two steps, now we can build the actual client side that handles incoming messages. We can use the next method `CvOp(CvUpdate)` to pass the CvUpdate as a closure so that the server side can apply the defined function on their states, in order to gain benefits from high availability since they are directly sent to the closest available server. The developer can also send `TOp(TUpdate)` to make all the replicas update their logs synchronously. Understanding the relation and differences between these two operation types can help developers improve the performance of the implementation as well as achieve certain consistency levels.

We use actors to represent different nodes or replicas in the cluster, and since actors use messages for expressing concurrent computation, all these operations can be treated as 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. The following interfaces are defined in Akka style. The client actor which connects to the protocol layer, can be implemented by extending the `Protocol` class, thereby inheriting the internal operation definitions and message handlers. Self-defined operations that need to be processed by the system are imported from the CounterCRDT object defined previously and can be sent using the following message handler in Listing 2.

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

The protocol implementation is explained in Section 4.1, and we use the same message handler on the client-side.

2.4. System structure

After the resetable counter example, let us have a look at the general design of the system in Fig. 2. There are three layers from bottom to top: replicas, distributed protocol, and application.

We already explained how applications interact with the protocol using the provided APIs. Now towards the storage layer, we skip the discussion about types of data storage but only focus on the data format. The server-side stores a log which contains CvRDT states and the sequence of TOps. The log might look like in Fig. 3.

```
class CounterClient extends Protocol[RGCounter] {

  import CounterCRDT._

  val CounterClientBehavior: Receive = {
    case Incr  => self forward CvOp(incr)
    case Get   => self forward TOp("Get")
    case Reset => self forward TOp("Reset")
    ...
  }
  override def receive =
    CounterClientBehavior.orElse(super.receive)
  ...
}
```

Listing 2: Client-side message handler for a resettable counter.

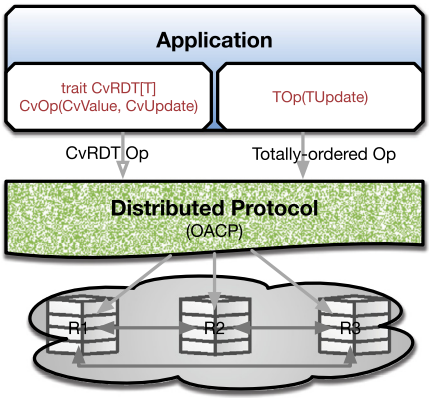

Fig. 2. High-level view of OACP.

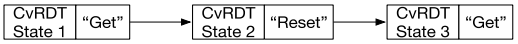

Fig. 3. Log abstraction in OACP.

The log does not record submitted CvOps, and only when a TOp is executed, the current CvRDT state is stored together with the TOp label in the log entry.

3. Observable atomic consistency

3.1. Defining observable atomic consistency

The definition of observable atomic consistency requires the definition of partial order between operations, and the specification on how to apply all the operations on each site.

Definition 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 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 $p \prec q \vee q \prec p$

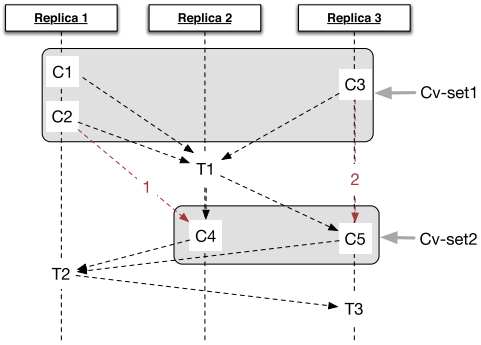

Fig. 4. CvT order of operations.

In Fig. 4, the partial order is indicated using black dashed arrows, while lines 1 and 2 (red dashed arrows) are derived from the transitivity of $\prec$. There are two different Cv-sets in this case, while the CvRDT updates within have no partial-order relationship.

In a CvT order, only the operations in the same Cv-set could be executed concurrently. Now we consider operations executed on different sites. Suppose each site i executes a linear extension[3] compatible with the CvT order. Then, the replicated system with n sites provides local atomic consistency, which is defined as follows.

Definition 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.

Definition 4 *(Observable atomic consistency (OAC)).* 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 three replicas in Fig. 4 could execute operations using different linear extensions of the CvT order. However, despite possible re-orderings, we need to see whether state convergence is guaranteed so that all replicas eventually reach the same state after performing the same set of operations.

3.2. State convergence

Definition 5 *(State convergence).* A LAC system is state convergent if all linear extensions of the underlying CvT order O reach the same state S.

Theorem 1. *Given a CvT order, if all operations in each Cv-set are commutative, then any LAC system is state convergent.*

In order to give a complete proof for Theorem 1, we first introduce the following lemmas and their proofs.

Lemma 1. *Given a legal serialization $O_i = (U, <_i)$ of CvT order $O = (U, \prec)$, if $\exists u, v \in U$ such that $u <_i v$ and $u \not\prec v$, then $\exists! C_i \in U$ such that $u \in C_i \wedge v \in C_i$.*

Proof. Let's consider two operations in the following cases and find which situation satisfies the requirement.

Case 1: if $u \in T$ and $v \in T$, $u <_i v$ then according to the definition, $u \prec v$.
Case 2: if $u \in T$ and $v \in C$, $u <_i v$, there must $\exists C_i$ such that $v \in C_i$, since $u \prec \{\forall c | c \in C_i\}$, $u \prec v$.
Case 3: if $u \in C$ and $v \in T$, $u <_i v$, using the same argument from Case 2, we have $u \prec v$.
Case 4: if $u \in C$ and $v \in C$, $u \in C_i \wedge v \in C_j \wedge C_i \cap C_j = \varnothing$, $u <_i v$ so that $C_i \prec C_j$, then we have $u \prec v$.
Case 5: if $u \in C_i$ and $v \in C_i$, $u \in C_i \wedge v \in C_i$, we know from the definition that $u \not\prec v$.

Consider the above five cases, only case 5 can satisfy $u <_i v \wedge u \not\prec v$, so the lemma is correct. □

[3] A linear extension of a partially ordered set P is a permutation of the elements p_1, p_2, ... of P such that $p_i < p_j$ implies $i < j$.

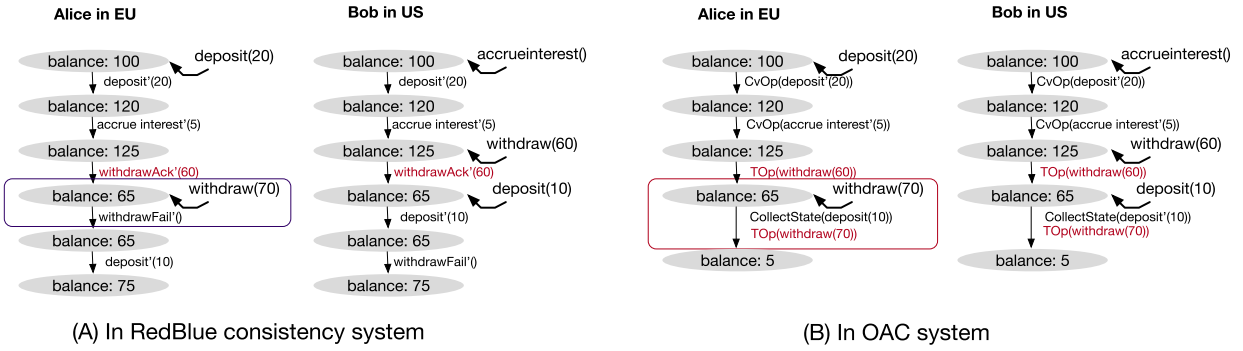

Fig. 5. Comparison between RedBlue consistency and OAC.

Lemma 2. *Assume $O_i = (U, <_i)$ and $O_j = (U, <_j)$ are both legal serializations of CvT order $O = (U, \prec)$ that are identical except for two adjacent operations a and b such that $a <_i b$ and $b <_j a$ and that all operations $c \in C$ are commutative inside each single Cv-set. If S is the initial state, then $S(O_i) = S(O_j)$.*

Proof for lemma. Let P and Q be the greatest common prefix and suffix of O_i and O_j, then $S_{ab} = S(P) + a + b$, $S_{ba} = S(P) + b + a$.

From Lemma 1, we know that only the CvRDT operations inside the same Cv-set can be the two adjacent operations a and b. The operation in one Cv-set commute, so $S_{ab} = S_{ba}$. And we could also derive that $S_{ab}(Q) = S_{ba}(Q)$. Since $S_{ab}(Q) = S(O_i)$ and $S_{ba}(Q) = S_(O_j)$ then we have $S(O_i) = S(O_j)$. □

Now we come back to the proof of Theorem 1.

Proof for theorem. Considering when:

(1) $O_i = O_j$. Then we have $S(O_i) = S(O_j)$. (2) $O_i \neq O_j$, then there exists at least one pair of adjacent operations such as u and v that $u <_i v$ and $v <_j u$. From Lemma 2 we know if we swap the sequence of u and v in O_i, we could get a new execution sequence O_{i+1} such that $S(O_{i+1}) = S(O_i)$. If $O_{i+1} \neq O_j$, then we continue to find adjacent operation pairs in O_{i+1} until we finally construct a sequence of operations which is equal to O_j.

Then we proved that any LAC system is state convergent. □

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

3.3. More discussions

Comparison to RedBlue Consistency. A closely related consistency model is RedBlue consistency [15]. Red operations are the ones that 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. The applications are also required to adjust to using these shadow operations. The shadow operations can "introduce some surprising anomalies to a user experience" [15]. 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 global 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.

Here is an example for comparison. There are Alice in the EU and Bob in the US, performing some operations on the same account. Suppose the sequence of bank operations arriving at any of the replicas is as follows: Alice(deposit(20)) → Bob(accrueinterest()) → Bob(withdraw(60)) → Bob(deposit(10)) → Alice(withdraw(70)).

RedBlue consistency requires the definition of withdrawAck' and withdrawFail' as shadow operations for an original "withdraw" operation. Then, withdrawAck' is marked as a red operation (i.e., totally ordered).

In RedBlue consistency, when Alice wants to execute withdraw(70) (blue frame in Fig. 5(A)), there is an immediate response withdrawFail'() since it is a blue operation. In the local view of Alice, the deposit(10) from Bob is not visible yet. However, in the assumed arriving sequence of operations, deposit(10) from Bob was already received by one of the replicas; thus, in principle, the withdraw could have succeeded.

Compared with RedBlue consistency, in the OAC system, the evolution of the system state looks as shown in Fig. 5(B). We only need to define which operations should be CvOps and which should be TOps. Meanwhile, when Alice wants to execute withdraw(70) since it is a TOp, the system will make all the replicas reach the same state to make sure Alice has a consistent global view of the system (red frame in Fig. 5(B)).

In RedBlue consistent system, there are three rules we can apply to decide which shadow operations can be blue or must be red. If now we extend the bank system with a read operation to show the accurate account balance to the user, this operation is not commutative with all the other operations, and according to the first rule, "For any pair of non-commutative shadow operations u and v, label both u and v red", we would need to label all the existing operations as red, which means operations with strong synchronization. In contrast, using OAC, the "deposit" and "accrueinterest" operations can still be CvOps, which are more efficient than red operations in RedBlue consistency.

4. Observable atomic consistency protocol

Following the definition of observable atomic consistency (see Definition 4), 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 [16] and the *system model* where we describe the assumptions; second, we describe the protocol from the client-side and the server-side; third, we discuss the fault model for our protocol and some implementation details; finally, we discuss differences to the most closely related protocol.

Data model. The data model is given by the function $\texttt{OACPVal} : \texttt{Read} \times \{\texttt{CvUpdate}, \texttt{TUpdate}\}^* \rightarrow \texttt{Value}$. This is a function that 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 abstract operations: `CvUpdate` represents CvRDT updates, `TUpdate` represents totally-ordered updates, and `Read` represents totally-ordered read operations.

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.

System model. The system model that we use is a client-server model, where multiple servers can respond to the clients. As networking abstraction we assume asynchronous communication that messages are never lost, and a server node can fail and then come back to the cluster. Under this assumption, our system with n servers can tolerate $(n/2 - 1)$ faults.

4.1. OACP protocol

OACP client-side protocol. The OACP client-side protocol is rather simple, an example program can be found in Section 2.2. 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 [17]). Thus, we omit the pseudocode here.

An important assumption of the protocol is that the same client never invokes an operation before the previous invocation gets a response. In particular, when a TOp happens directly after a CvOp, the client awaits an acknowledgment of the previous message from the server-side (which only requires reaching one available replica) before invoking the TOp, and in this way the program order on the client-side is preserved.

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 online chatting and Twitter-like micro-blogging.

In Listing 3, we use `onReceive` message handler to describe the behavior of the server when specific messages are received.

When the server receives a `CvOp` (line 8), it first checks the special flag "frozen". This flag is set to true during the processing of a TOp, which prevents subsequent operations from interfering with the consensus stage. Moreover, when the flag is true, the following operations will be stashed in a buffer for later processing. If the flag is false then the server merges the current CvRDT state (using `merge`), and an acknowledgment is sent to the client. The update is broadcast to the other servers (`Broadcast`) in the background. If a received update is from the server-side, the server updates its local state (line 18).

TOps are, by assumption, only processed by the leader server. If a follower server receives a TOp, it will forward the TOp to the current leader server. When a leader server receives a TOp (either a TUpdate or a consistent Read), the frozen flag

```
1  role OACP_Server {
2    var currentState: CRDT;
3    var up: Update;
4    var log: Entry*;
5    var currentLeader: Server;
6    var client: Client;
7    var frozen: Boolean;
8    var isLeader: Boolean;
9
10   onReceive(msg: CvOp) {
11     if frozen then { stash(msg); }
12     else if msg.sender.type == Client then {
13       currentState = currentState.merge(msg.CvUpdate);
14       msg.sender.reply(); // acknowledge to client
15       BroadcastInBackground(msg); // send update to other replicas
16     }
17     else // receive the broadcast
18       currentState = currentState.merge(msg.CvUpdate);
19   }
20
21   onReceive(msg: TOp) {
22     if isLeader && frozen then stash(msg);
23     else if isLeader then {
24       frozen = true;
25       numStateMsgReceived = 0;
26       client = msg.sender;
27       up = msg.(TUpdate | Read);
28       BroadcastInBackground(GetState);
29     }
30     else forward(currentLeader, msg);
31   }
32
33   onReceive(msg: GetState) {
34     frozen = true;
35     reply(StateIs(currentState));
36   }
37
38   onReceive(msg: StateIs) {
39     if isLeader then {
40       numStateMsgReceived += 1;
41       currentState = currentState.merge(msg.cState);
42       if numStateMsgReceived >= numReplicas/2+1 then {
43         RTOB(new Entry {
44                origin = msg.sender, number = log.nextIndex,
45                cState = currentState, toUpdate = up });
46         log = log ++ e;
47         client.reply(OACPVal(updates(log)));
48         BroadcastInBackground(Melt);
49       }
50     }
51   }
52
53   onReceive(e: Entry) {
54     log = log ++ e;
55   }
56
57   onReceive(msg: Melt) {
58     frozen = false;
59     unstash();
60   }
61
62   function updates(l: Log): (TUpdate*, cState) =
63     return (l[0].toUpdate ++ ... ++ l[l.length-1].toUpdate, l[l.length-1].cState);
64 }
```

Listing 3: Server-side OACP pseudocode (unoptimized).

needs to be checked similarly as before (line 22). If the system is available, the frozen flag is set to true, and the leader server broadcasts a `GetState` message (line 23) to collect the current states from the other replicas.

All nodes that receive the message `GetState` set their frozen flag to true and reply with their current CRDT states (lines 33–36). If the number of replies reaches a majority of the replicas, then we perform an RTOB with a new entry containing

the previous `TOp` (lines 42–45). The leader node will append the entry to the current log (line 46) and meanwhile, a response with the current result is sent to the client (line 47).

Once a server receives an entry through RTOB, the server will update its state by appending the entry to the current log (line 54).

The leader server broadcasts a `Melt` message to reset the "frozen" flag when the RTOB process is finished (lines 48 and 58). Then, previously stashed messages are put back into the message queue (line 59).

In our OACP reference implementation, RTOB is implemented using the Raft consensus protocol. When the `TOp` is handled by a non-leader server, the server forwards the update to the current leader (line 30). Moreover, we define `Read` as a TOp to guarantee strong consistency.

When the leader receives acknowledgments from a majority of servers, a "Melt" message (line 48) is broadcast to reset the "frozen" flag to false. If there are n servers in the cluster, then there are $(n-1)$ messages added to the messages exchanged by the protocol. Therefore, we devised an optimized version (see Listing 4) to reset the flag to false each time a server makes a commit in the consensus protocol.

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

Listing 4: Server-side OACP pseudocode (optimized).

4.2. More discussions

Fault Model. The fault model of the OACP system consists of two parts. (1) During the processing of an RTOB, crash failures of n nodes are tolerated in a cluster with $2n+1$ nodes using Raft. (2) During the CRDT phase, since all the server broadcast the updates they receive from the client, and the messages are never lost, then the system can guarantee the eventual delivery of the CvUpdates. In this case, we only need to collect the CRDT states from a majority of replicas. Thus, crash failures of n nodes are tolerated in a cluster with $2n+1$ nodes, the same as when using Raft.

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.

CRDT Implementation. When we implement the CRDTs in our protocol, there is a gossiping mechanism that is not shown in Listing 3. The servers periodically send their entire CRDT states to the other nodes in the cluster and merge the states they receive from the other nodes.

OACPVal. `OACPVal` is defined according to the concrete applications. For example, in our previous resettable-counter example in Section 2.3, it can be defined as in Listing 5.

```
1  case object RGCounterVal {
2    override def OACPVal(log: List[Entry[Array[Int], String]]) = {
3      var recent = 0
4      log foreach {
5        item => item.NMCommand.get match {
6          case "Reset" => recent = item.index
7        }
8      }
9      value(diff(log(log.size).mState.get,log(recent).mState.get))
10   }
11 }
```

Listing 5: Source code for OACPVal in a resettable counter.

Where `value` computes the actual number of a GCounter and `diff` computes the difference between two versions of GCounters, we omit their implementation code for simplicity.

Comparison to GSP. GSP [16,18] 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.

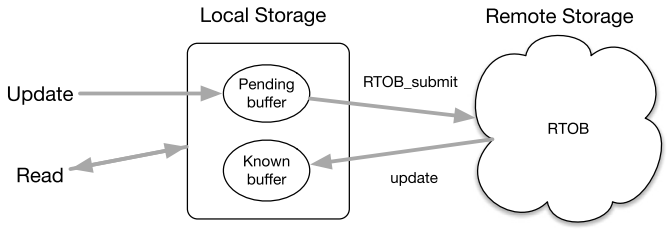

Fig. 6. GSP protocol abstraction.

Fig. 6 shows the abstraction of GSP protocol.

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 anomaly conditions such as the following from paper [16]:

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

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 $wr(B, 1)$.

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 the read operations, and all the operations in GSP will need RTOB, which has a high cost.

5. Formal model and model checking

In this section, we use Maude [9] to help verify some properties of OACP. First, we give an overview of Maude and then explain how we use Maude to model OACP. Finally, we present the verification result so that we have more confidence that OACP guarantees a consistent view of convergent state once a TOp is executed, and the CvT order (see Section 3) is preserved during execution.

5.1. Abstraction

We care about the consistency of the different replicas but are not interested in the actual content, so we abstract from it. The implementation of OACP requires a reliable total order broadcast primitive for which we use the Raft consensus protocol. However, Raft requires a random timeout mechanism for leader election. We could use timers to model this, but it would increase the complexity of searching for all the possible results. Thus, we use the same idea in [19] and consider a system without heartbeats, and where only one candidate is possible at a time. If the leader is unavailable, it will hand the leadership to the next neighbor. We also assume that unavailable nodes (a) can still respond negatively to messages and (b) do not update their states.

5.2. Assumptions

We assume the communication in the OACP model is unordered; each node that is selected as a leader knows about all the other nodes; messages can be resent but are never lost; replicas might crash and then recover.

5.3. Maude overview

Distributed systems are dynamic. The outcome of the same sequence of operations might not always be the same. Maude is an ideal modeling tool here, because the rewriting logic [20] it uses naturally reflects the state change of the system in a non-deterministic way. More tutorials and examples can be found in the Maude manual and book "Designing Reliable Distributed Systems [21]". Here we select some of the key concepts for a brief illustration.

Module. A module in Maude is defined using the following syntax:

```
fmod MODULENAME is BODY
endfm
```

Within the BODY, the syntax and semantics of the module are defined. The syntax includes data types and function names. For example, sort Nat . is a natural number type and op $+$: Nat Nat- >Nat . defines an operator "+" which takes two Nat and returns a resulting Nat.

The semantics include equations and rules for modifying terms. Equations are used for defining how to reduce a term. They are defined by writing some left-hand side terms equal some right-hand side terms. For example, eq $0 + M = M$. and eq $s(M) + N = s(M + N)$. define the addition function recursively. And when we compare two terms such as $s(s(0)) + s(0) = s(s(s(0))) + 0$ (i.e. 2+1=3+0), we substitute the terms on each side until no more equations can be applied.

Rules are also used for describing how to rewrite a term. However, they are not intended to capture equality in a system. Instead, they capture transitions between states. Rewriting logic is explained in the following paragraph.

Rewriting logic. Rewrite rules describe how a term evolves to the next state. The rules can have labels such as $l : t \to t'$. If it is a conditional rule, then the form is like $l : t \to t'$ **if** *cond*. Since different rules can be applied to the same state, we are able to model non-deterministic behavior. Real code in Maude looks like the following:

```
rl[l] : t=>t' .
```

and

```
crl[l] : t=>t' if cond.
```

Search. Maude allows us to execute an evaluation step-by-step using rew and frew commands, but for state-space exploration, the search command can be used to go through all behaviors from the initial state.

Asynchronous communication. Communication in OACP is asynchronous, and all the operations are treated as messages. It is essential to understand how to model asynchronous communication in Maude.

We use message wrappers to represent "send content from a sender to a receiver" using the following syntax:

```
op msg_from_to_ : MsgContent Oid Oid -> Msg [ctor] .
```

Using the same idea, we use a multicast wrapper to represent "send content from a sender to receivers 1, 2, 3, ..." using the following syntax:

```
op multicast_from_to_ : MsgContent Oid OidSet -> Msg [ctor] .
```

Since we assume that each node that will be selected as a leader will know about all the other nodes, we can use multicast to all the other nodes to express the concept of the broadcast.

Configuration. Another useful pattern is to model concurrent objects. The CONFIGURATION module is pre-defined in Maude in order to support using a term to represent a certain state of an object. The syntax is the following:

```
op <_: C | att1:_, ..., attn:_> : Oid s1 ... sn -> Object [ctor] .
```

The CONFIGURATION module allows us to describe the concurrent modification of the object. We use it to describe the state of nodes in the following sections.

5.4. Modeling

Our OACP model in Maude[4] is based on a previous model for the Raft protocol. Since the Raft protocol is not the main interest of our paper, we do not include a detailed discussion here. We extend the specification with a CRDT module, extend the message handlers, and modify the message handling behaviors of the nodes. In the following, we mention some interesting modules together with the key implementations.

[4] Our open-source Maude implementation can be found on GitHub; see [22].

GSET_NAT module. We defined a simple CRDT type: grow-only set (GSet) of natural numbers and include some essential operations in this module. The Grow-only set only allows "add" operations, and one the item added, cannot be removed. Merging two GSets is to compute the union of the two sets. The purpose of using a simple data type is to simplify the verification difficulty in Maude, and the example is representative enough for more general cases.

LEDGER module. This module defines the structure of logs that are described in the Raft protocol. Each log consists of multiple entries. Based on the original Raft log structure, a GSet state is added for each entry in order to keep a record of the convergent state (see Fig. 3). In this way, we could use the comparison between logs on different nodes to check whether the convergent state is the same once the TOp is executed.

NODE module. These modules define the behavior of the nodes. We extend the attributes in module NODE with crdtState, freezewaitingfor and some related boolean flags, including frozen and freezewaiting. The freezewaitingfor is used to guarantee the scheduling order of the operations. The frozen flag is for checking whether there is a TOp processed by the system, and thus CvOps are not executed. The freezewaiting flag is used to guarantee that the system state will not change until all the responses are received. When we verify the ordering relations between different operations, we add more attributes such as command − order, which will be described in Section 5.5.3.

MESSAGE module. Different types of messages are defined in module MESSAGE. On top of the essential Raft messages, we add a special message Freeze. Freeze and Query are executed as a bundle in order to enable the shifting between handling CvOps and TOps. For TOps, the message will be passed around by the Raft protocols, while for CvOps, the message can be processed concurrently by any node and propagate eventually to every node. Freeze messages are intended to set the frozen flag on each node to true and then collect all the current CRDT state on each node. After the commit phase for all the nodes, the frozen flag will be set back to false to enable concurrent message handling again.

LEADER, FOLLOWER and CANDIDATE module. These modules define node behaviors. For CvOps message handlers, all the nodes have the same behavior. The frozen flag is checked in order not to process concurrent messages when the system is running the Raft consensus protocol (see Listing 6 Line 4). The node which receives the update message notifies the other nodes in the cluster to also change their state (see Listing 6 Line 7). To simplify the modeling for gossiping behaviors, the update message will be only sent once to each neighbor node, so when the multicasting happens, the updateBar messages are sent to distinguish between the sender and avoid endless state exchanging.

```
1  crl [update-crdt-*] :
2    (msg UpdateCRDT(up) from cli to fol)
3    < fol : Node | crdtState : CRDT,        neighbors : fols,
4           frozen : false, AS > =>
5    < fol : Node | crdtState : CRDT U {up}, neighbors : fols,
6           frozen : false, AS >
7    (multicast UpdateBarCRDT(up) from fol to fols) .
8
9  crl [updatebar-crdt-*-1] :
10   (msg UpdateBarCRDT(up) from fol to lea)
11   < lea : Node | crdtState : CRDT,        frozen : false, AS > =>
12   < lea : Node | crdtState : CRDT U {up}, frozen : false, AS >
13   if not appears(CL) .
```

Listing 6: Common rules for all nodes.

The module LEADER contains a few more special rules for handling the response messages from followers. For example, in rule [freeze-leader], after the Freeze message reaches the leader node, the leader broadcast the messages to all the other nodes (see Listing 7 Line 7) and waits for the responses from all the followers. The frozen flag set from false to true (see Listing 7 Line 3 and 5) to block the synchronize CvOp messages. The freezewaiting flag set from false to true (see Listing 7 Line 4 and 6) to collect all the convergent state on the other nodes. The freezewaitingfor field set as C to guarantee the Query(C) will be scheduled next instead of other commands. The updating of the state will happen in Listing 7 Line 13. After receiving all the responses, the CRDT state will be the same on each node and will be logged into the leader's entry. Thus it is ready for accepting a Query message here (which can be considered as a TOp). In [query-leader] rule, the leader node puts the current CRDT state together with all the other attributes into the entry and starts Raft protocol.

OFFLINE module. For nodes that are offline, we made the rule that all the messages received by the node will be forward to one of its neighbor nodes. Furthermore, all the decisions made by offline nodes will be passive, and the current state of the nodes should not change, including the flags.

```
rl [freeze-leader] :
   (msg Freeze(C) from cli to lea)
   < lea : LeaderNode | neighbors : fols, crdtState : CRDT, frozen : false,
          freezewaiting : false, freezewaiting : "", AS > =>
   < lea : LeaderNode | neighbors : fols, crdtState : CRDT, frozen : true,
          freezewaiting : true,  freezewaiting : C,  AS >
   (multicast Freeze(C) from lea to fols) .

rl [freeze-response-leader] :
  (msg FreezeResponse(gset) from fol to lea)
  < lea : LeaderNode | crdtState : CRDT,        frozen : true,
        freezewaiting : true, number-response : N2,     AS > =>
  < lea : LeaderNode | crdtState : CRDT U gset, frozen : true,
        freezewaiting : true, number-response : N2 + 1, AS > .

rl [freeze-done-leader] :
  < lea : LeaderNode | frozen : true, freezewaiting : true,
        number-response : N3, number-neighbors : N3, AS > =>
  < lea : LeaderNode | frozen : true, freezewaiting : false,
        number-response : N3, number-neighbors : N3, AS > .

crl [query-leader] :
  (msg Query(C) from cli to lea)
  < lea : LeaderNode | currentTerm : term, log : led , neighbors : fols,
        crdtState : CRDT, frozen : true, waiting : false,
        freezewaiting : false, freezewaitingfor: C, number-yes : N1,
        number-response : N2,
        AS, command-order : CL, AS >
        =>
  < lea : LeaderNode | currentTerm : term, log : led', neighbors : fols,
        crdtState : CRDT, frozen : true, waiting : true ,
        freezewaiting : false, freezewaitingfor: C, number-yes : 1 ,
        number-response : 0 ,
        command-order : CL ; C, AS >
  (multicast SetLog(term, led') from lea to fols)
  if led' := led ; entry(index(head(led)) + 1, term, C, CRDT) .
```

Listing 7: Special rules for the leader node.

5.5. Verification

5.5.1. A simple test case

In our verification step, we focus on one specific simple test case which covers all the features of OACP. The test case is defined in Listing 8.

```
op init-client : Nat Nat -> Configuration .
eq init-client(s N, g) =
  (msg BecomeLeader(2) from client to node(0))
  (msg Freeze("cmd1", empty) from client to node(0))
  (msg Query("cmd1") from client to node(0))
  (msg UpdateCRDT(1, "update1", "cmd1", "cmd2") from client to node(1))
  (msg UpdateCRDT(2, "update2", "cmd1", "cmd2") from client to node(2))
  (msg Freeze("cmd2", "update1" ; "update2") from client to node(0))
  (msg Query("cmd2") from client to node(0))
  (msg UpdateCRDT(3, "update3", "cmd2", "cmd3") from client to node(1))
  (msg Freeze("cmd3", "update3") from client to node(0))
  (msg Query("cmd3") from client to node(0))
  < client : Client | gas : g, num : get-minority(s N),
      live : make-neighbors(s N, N), fail : none > .
```

Listing 8: Test case for Maude.

CRDT states in the entries of each committed log are consistent on all the nodes so that the protocol can guarantee observable atomic consistency. A Query message always comes together with Freeze as we explain above, and the leader will commit the log entry during the execution of these operations.

The test case combines several concurrent UpdateCRDT messages (CvOps) with multiple Query messages (TOps) to the system, which is a small but complete example to test whether the log is consistent including the convergent state whenever a TOp is executed.

SIMUL module. This module defines a client and simulates sending requests to the system. Node(0) first gets elected as the leader after the initialization of the client, and then the client sends different queries to the cluster.

5.5.2. Log consistency and state convergence

One of the most important properties of a consistency protocol is whether the protocol provides consistent results. Note that the CRDT state in the configuration of each node is not always consistent due to the asynchronous execution. Thus, we cannot compare the CRDT states directly. Luckily, OACP provides us with observable atomic consistency, which means that all the states in a committed log entry should be consistent with others.

Using the following command, we can check whether there is a state where the CRDT state in a committed log entry is not consistent with others. If there is a solution for that, it means the protocol cannot provide strong consistency for TOps.

```
search [1] init(3, 1) =>* C:Configuration
                      < O1:Oid : C1:Cid | committed : L11:Ledger ;
                      entry(ind:Nat, term:Nat, C:Command, cState1:GSet) ;
                      L12:Ledger, AS1:AttributeSet >
                      < O2:Oid : C2:Cid | committed : L21:Ledger ;
                      entry(ind:Nat, term:Nat, C:Command, cState2:GSet) ;
                      L22:Ledger, AS2:AttributeSet >
                      such that cState1:GSet =/= cState2:GSet
                             or L11:Ledger =/= L21:Ledger .
```

Here, [1] means the search stops once a satisfying solution is found. The arrow `=>*` means we are searching a reachable state in one or more steps starting from the initial state. There is no solution after running the simulation, which means the protocol provides strong consistency for TOps while enabling concurrent updates via CvOps. The simulation performed by Maude is exhaustive and covers all possible executions given the initial configurations of the replicas and the client. The simulation of the test case takes about 730 s on a 4-core computer with 8 GB RAM and explores 94,338 states with 40,882,819 rewrites.

5.5.3. Order analysis

Another property we would like to verify is that for the test case in Listing 8, the order in which the operations arrive at each replica preserves the CvT order. Since Maude schedules messages in a random manner, it is necessary to provide more information about the order between messages. Our solution is to provide a partial order relation as an additional parameter in the message field.

In this case, we need to make another extension for the current model to be able to describe orders between operations.

The first extension is for the MESSAGE module. The message field is extended with two command lists. One is a list of commands that are before the current command according to the program order, and the other one is a list of commands that are after the current command according to the program order.

The second extension is for the message handling rules. First, we define a list of commands to describe the ordering relationship. For rules such as [query-leader] and [update-crdt-$*$] we extend the attributes with command-order to record the commands that are processed by each node. Next, we need to modify the rules so that one can combine the information kept in command-order and the message field to decide which message should be scheduled next.

For example, in the test case in Listing 8, the update CRDT message is defined as UpdateCRDT(1, "udpate1", "cmd1", "cmd2") which means it will update the current CRDT state to 1, its own command name is update1, it will be executed after cmd1 but before cmd2. In the new rule [update-crdt-$*$], we have an additional condition if appears(CL, CS1) where appears returns a Boolean indicating whether the elements in CS1 all appear in CL. This guarantees the UpdateCRDT is executed after cmd1.

```
crl [update-crdt-*] :
  (msg UpdateCRDT(up, C1, CS1, CS2) from cli to lea)
  < lea : Node | crdtState : CRDT,    neighbors : fols, frozen : false,
  command-order : CL,      AS > =>
  < lea : Node | crdtState : CRDT up, neighbors : fols, frozen : false,
  command-order : CL ; C1, AS >
  (multicast UpdateBarCRDT(up, C1, CS2) from lea to fols)
  if appears(CL, CS1) .
```

We can now make use of the command-order list that we used for keeping track of the order of different operations. The exclamation point in `=>!` indicates that only final states should be considered, so in our specification, we only list all the possible execution orders in the final state according to the definition of the CvT order and ask Maude to find a possible execution that has a different order. Since a node might fail right after it receives a CRDT update, we assume that CRDT updates might get lost.

There is no solution after running the simulation, which means the protocol guarantees the CvT order for each replica. The simulation of the test cases takes about 618,235 ms on a 4-core computer with 8 GB RAM and explores 90,507 states with 23,646,525 rewrites.)

```
search [1] init(3, 1) =>! C:Configuration
                      < O1:Oid : C1:Cid | command-order : CO1:command-list,
                      AS1:AttributeSet >
                      such that not contains(CO1:command-list, "cmd1" ;
                      "update1" ; "update2" ; "cmd2" ; "update3" ; "cmd3")
                      and not contains(CO1:command-list, "cmd1" ; "update2" ;
                      "update1" ; "cmd2" ; "update3" ; "cmd3")
                      and not contains(CO1:command-list, "cmd1" ; "update1" ;
                      "cmd2" ; "update3" ; "cmd3")
                      and not contains(CO1:command-list, "cmd1" ; "update2" ;
                      "cmd2" ; "update3" ; "cmd3")
                      and not contains(CO1:command-list, "cmd1" ; "update1" ;
                      "update2" ; "cmd2" ; "cmd3")
                      and not contains(CO1:command-list, "cmd1" ; "update2" ;
                      "update1" ; "cmd2" ; "cmd3")
                      and not contains(CO1:command-list, "cmd1" ; "update1" ;
                      "cmd2" ; "cmd3")
                      and not contains(CO1:command-list, "cmd1" ; "update2" ;
                      "cmd2" ; "cmd3") and
                      not contains(CO1:command-list, "cmd1" ; "cmd2" ; "cmd3") .
```

Listing 9: Listing for the second verification.

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

Fig. 7. Average round-trip latency between sites on Amazon EC2.

5.5.4. Message usage

The last property we verify using Maude is message consumption. It is required that rewriting continues as long as there is a message in the message soup. We want to make sure that all messages that are generated during protocol execution are properly consumed. Thus, we use the following specification, trying to find a case where there are still messages in the message soup when the execution finishes and returns no solution.

```
search [1] in SIMUL : init(3, 1) =>! C:Configuration  M:Msg .
```

For the above two properties that we verified using Maude, we can only say that OACP works correctly for this specific test case. However, this model checking verification has significantly increased our confidence that OACP is correct.

6. 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 [23].

Experimental set-up. Our OACP implementation is based on the cluster extension of the widely-used Akka [7] actor framework. 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 an Amazon EC2 cluster which includes three T2 micro instances (1 vCPU, 1 GB memory, and EBS-only storage) running in Ohio, London, and Sydney. Fig. 7 shows the average round-trip latency between each pair of sites. Our implementation, available open-source [8], 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 when 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 Fig. 8. The cluster in our experiments consists of three nodes which locate in different regions. A leader node needs to be elected to keep the consensus. We put the client node in London and measure different conditions when the leader node is located 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 is located in Sydney, the maximum latency is 600 ms. Moreover, when the leader node is located in the other two regions, the maximum latency is around 350 ms.

Throughput. Now we focus on the throughput of OACP. We generate benchmarks with different proportions of CvOps and TOps. While we increase the number of concurrent requests to the same consistent log in the cluster, we measure the

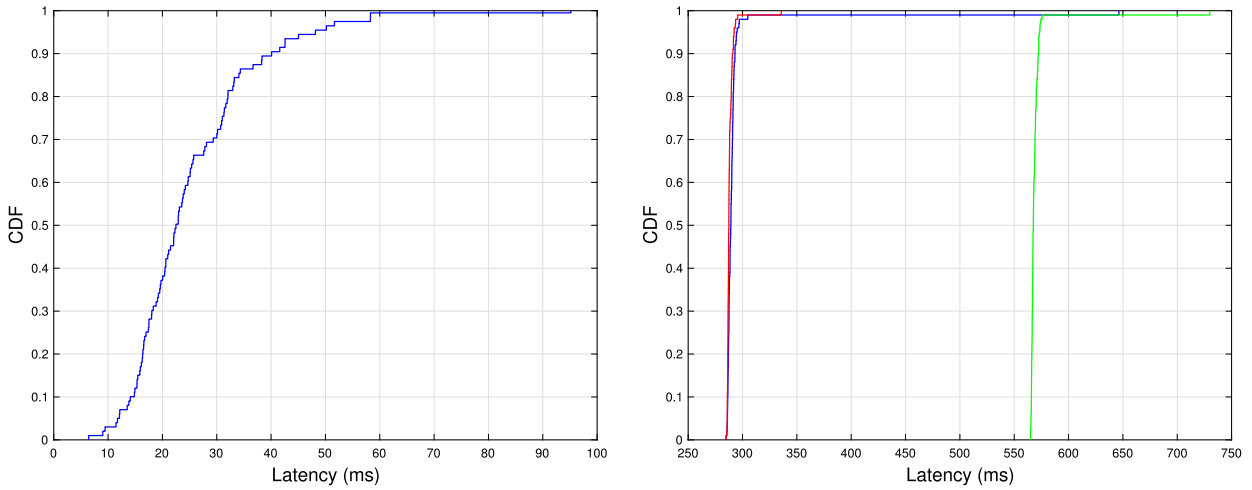

Fig. 8. Latency CDF for CvOps and TOps when the leader node is located in different regions. In (b), the green (the rightmost) line corresponds to the condition where the leader is located in Sydney, the blue line corresponds to Ohio, and the red (the leftmost) line corresponds to London. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

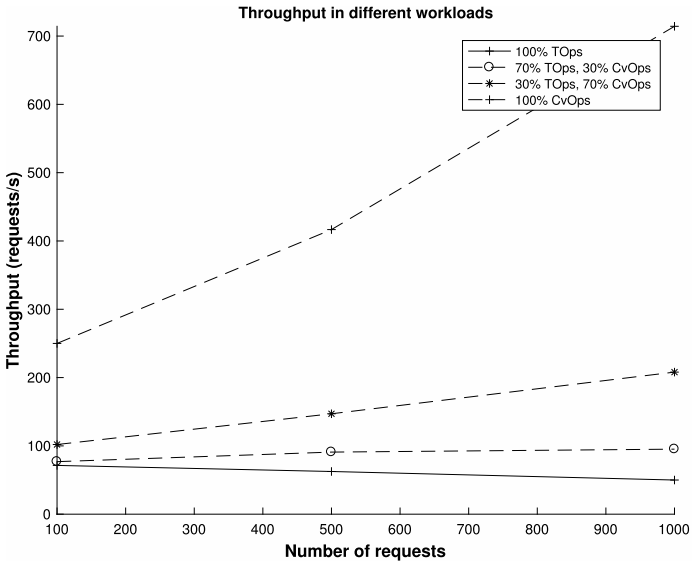

Fig. 9. Throughput for a 3-site cluster with varying CvOp and TOp workload mixes.

duration for processing all of the requests. The throughput is then the number of requests divided by the duration. The results in Fig. 9 show that increasing the ratio of TOps decreases the throughput. TOps require the leader node to force all the other replicas to reach a consensus on the same 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 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 [24]. 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 [25].

Scalability. To investigate scalability, we measure the number of exchanged messages in replicas of different sizes, ranging from 3 to 7 nodes. A Raft cluster is recommended with five nodes [17], and one with more than seven nodes is not common; thus, we do not consider them here.

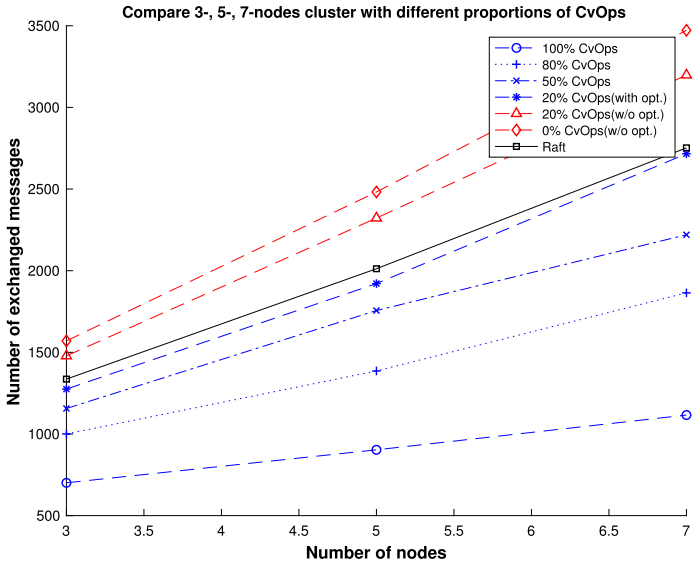

Fig. 10. Scalability comparison between Raft and OACP with different proportions of CvOps.

We create a small benchmark where a client sends 100 requests to the system. We change the proportion of CvOps in the benchmark and record the number of exchanged messages (excluding heartbeat messages for Raft protocol) and get the Fig. 10. It shows the scalability trend for Raft and OACP with different proportions of CvOps. As we can see, compared to Raft, the number of exchanged messages per request is less affected by the number of nodes in a cluster in OACP. Moreover, scalability is higher for cases with a higher percentage of CvOps.

6.1. Microbenchmark

We start the evaluation with a simple shopping cart benchmark to see the advantages and weaknesses of the OACP protocol. We use an Observed-Remove Set (ORSet) for implementing the shopping cart and 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 following two baseline protocols.

- Baseline protocol: every operation is submitted using RTOB to maintain consistency on all replicas; RTOB is implemented using the state-of-the-art Raft protocol [17].
- Baseline protocol with batching: an optimized version that provides a fixed batching buffer and allows us to submit multiple buffered operations at once.
- OACP: the protocol as described in Section 4.

We compared the number of exchanged messages among the baseline protocol, the batching protocol (batching buffer size: 5000 operations), and the OACP protocol, using a 3-node cluster. The results are shown in Fig. 11a.

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 Fig. 11a, 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 Fig. 11b, 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.

6.2. 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

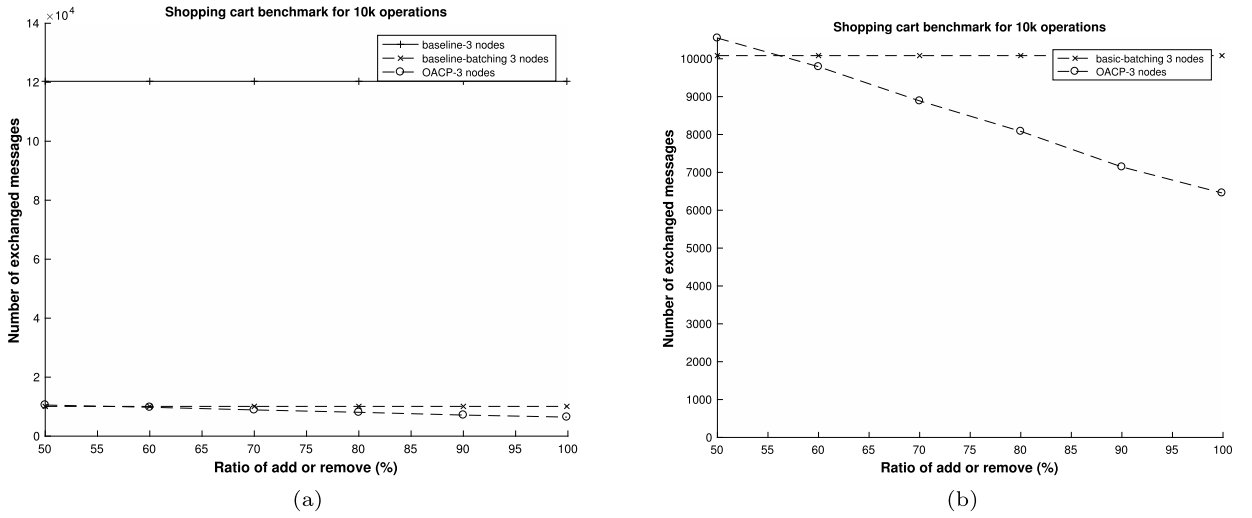

Fig. 11. Comparison on coordination among different protocols.

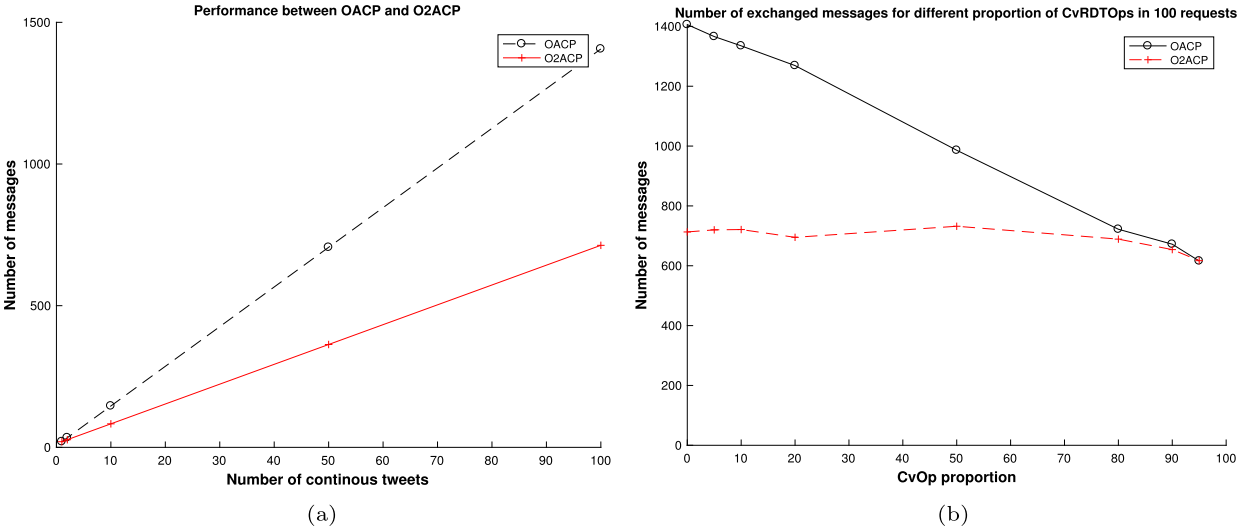

Fig. 12. OACP vs. O^2ACP.

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^2ACP.

In order to make a performance comparison to the original OACP, we generate random sequences of operations according to the proportion of CvOps and then execute through both protocols. The results are shown in Fig. 12a. When the proportion of tweets grows, the increase of messages is significantly smaller in the case of O^2ACP than in the case of OACP. In the case of 100 tweets, O^2ACP 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 Fig. 12b. 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^2ACP, since when CvOps increase from 0% to 95%, the exchanged messages in OACP reduce from around 1400 to 600, while in O^2ACP, the number of exchanged messages remains quite stable.

7. Related work

Distributed systems. Since the CAP theorem [1] establishes the impossibility to simultaneously achieve consistency, availability, and partition tolerance for any distributed system running on top of an asynchronous network, system implementations need to find trade-offs. Some are ensuring application correctness, e.g., Zookeeper [26] provides sequential consistency [27] such that updates from a client are applied in the order in which they were sent. Some are focusing on performance, e.g., Bayou [28] is designed to support real-time collaborative applications and thus gives up consistency for high availability, providing eventual consistency [29]. Some are more flexible such as Consistency Rationing [30], which allows a user to specify required consistency guarantees as well as switch the guarantees at runtime automatically.

System with mixed consistency. Consistency models are used to specify contracts between programmers and systems. As long as the programmer follows the rules, the system guarantees the memory consistency and return predictable results. Fork consistency [31,32] allows users to read from *forked sites* that may not be up-to-date. In contrast, write operations require all sites to be up-to-date. Explicit consistency in Indigo [33] guarantees the preservation of specific invariants to strengthen consistency beyond eventual consistency. Lazy replication [34] provides a solution to ensure high availability while keeping the data consistent. It divides updates into three categories: casual, forced, and immediate. Immediate operations are equivalent to our totally-ordered operations. RedBlue consistency [15] is closely related to observable atomic consistency; we provide a comparison at the end of Section 3. Global-Local view [35] also proposes the idea of separating the state into local fast states and distant shared states. SIEVE [36] is a system based on RedBlue consistency, which automatically chooses consistency levels according to a user's definition of system invariants.

Distributed application development frameworks. The Akka framework [7,37], which we use to implement OACP (see Section 6), provides a widely-used implementation of the actor model [38] on the JVM for writing highly concurrent, distributed, and resilient applications. There are several other distributed development frameworks related to OACP. Correctables [39] is an abstraction to decouple applications from their underlying database, which also provides incremental consistency guarantees to compose multiple consistency levels. QUELEA [40] enables programmers to provide fine-grained application-level consistency constraints, which are mapped automatically to the guarantees ensured by the underlying data store. GSP [16,18] 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. We provide a more detailed comparison of GSP and OACP in Section 4. Orleans [41] provides a *virtual actor* abstraction for modeling and implementing distributed systems.

Data management in distributed systems. Paxos [42] and Raft [17] are popular consensus protocols in replicated systems and serve as the basis for reliable total order broadcast [14]. The ZooKeeper Atomic Broadcast protocol [43] (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 [3] and cloud types [44,45] resolve conflicts automatically using convergent operations, but they require specific data structures to provide commutativity. Not all of the possible update operations are commutative, and so not all problems can be cast to CRDTs. Bloom [46], Lasp [47] and Hamsaz [48] are some programming models that handle concurrent conflicts as part of a shared object's implementation. Cassandra [49], CaCOPS [50], Eiger [51], and ChainReaction [52] use the last-write-wins strategy to ensure availability; however, they may lose data if concurrent writes happen frequently enough. Riak [4] and mergeable types [53] provide the ability to resolve write conflicts on the application level.

8. Conclusion and future work

8.1. Conclusion

We design, verify, and implement a novel observable atomic consistency protocol (OACP), which is an extension of CvRDTs with totally-ordered operations. There are two main benefits from the design: (a) the consistency guarantees for available applications are strengthened on demand and (b) anomalies caused by local reads (e.g., in GSP) are avoided. It is important to note that there is no perfect model for all possible scenarios–we can only evaluate trade-offs in certain scenarios and for different application requirements. We prove that OACP guarantees state convergence of replicas and verify log consistency for totally-ordered operations using model checking methods. We also implement the protocol using the Akka actor-based middleware and provide a user-level API on top of the underlying system. Finally, we experimentally evaluate latency and throughput properties using a geo-distributed cluster running on Amazon EC2. The main insights from our experimental results are that (a) using OACP, the higher the ratio of convergent operations, the lower the coordination overhead becomes compared to other protocols providing strong consistency, and (b) there is only a small overhead caused by the introduction of convergent operations.

8.2. Future work

Currently, our extension supports only state-based CRDTs whose entire state needs to be exchanged among replicas. This can become a bottleneck for more complex user-defined CRDTs. We want to explore the possibility for operation-based CRDTs which only propagate update operations, and thus make the protocol more general. We would also like to compare the performance difference between CRDTs from Akka distributed data and our implementation. Moreover, the current user API is pretty simple. Another research direction will be to provide a domain-specific language for a distributed system with more features.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is supported by the China Scholarship Council201600160040, and by the Tianhe Supercomputer Project 2018YFB0204301.

References

[1] S. Gilbert, N.A. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33 (2002) 51–59.
[2] M. Shapiro, N.M. Preguiça, C. Baquero, M. Zawirski, Conflict-Free Replicated Data Types, Springer Verlag, 2011, pp. 386–400.
[3] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, A comprehensive study of convergent and commutative replicated data types, Tech. Rep. RR-7506; inria-00555588, HAL CCSD, Jan. 2011.
[4] R. Brown, S. Cribbs, C. Meiklejohn, S. Elliott, Riak DT map: a composable, convergent replicated dictionary, in: PaPEC@EuroSys, ACM, 2014, p. 1:1.
[5] Basho Technologies, Inc., Riak dt source code repository, https://github.com/basho/riak_dt, 2012-2017.
[6] The SyncFree Consortium, AntidoteDB: a planet-scale, available, transactional database with strong semantics.
[7] Lightbend, Inc., Akka, http://akka.io/. (Accessed 20 March 2016).
[8] X. Zhao, OACP implementation source code repository, https://github.com/CynthiaZ92/OACP, 2018.
[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C.L. Talcott (Eds.), All About Maude - a High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, vol. 4350, Springer, 2007.
[10] X. Zhao, P. Haller, Observable atomic consistency for CvRDTs, in: Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2018, ACM, Boston, MA, USA, 2018, pp. 23–32.
[11] M. Odersky, L. Spoon, B. Venners, Programming in Scala, 3rd edition, Artima Press, Mountain View, CA, 2016.
[12] M. Odersky, M. Zenger, Scalable component abstractions, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA, 2005, pp. 41–57.
[13] C. Baquero, P.S. Almeida, C. Lerche, The problem with embedded crdt counters and a solution, in: PaPoC@EuroSys, ACM, 2016, pp. 10:1–10:3.
[14] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: taxonomy and survey, CSURV: Comput. Surv. 36 (2004) 372–421.
[15] C. Li, D. Porto, A. Clement, J. Gehrke, N.M. Preguiça, R. Rodrigues, Making geo-replicated systems fast as possible, consistent when necessary, in: OSDI, USENIX Association, 2012, pp. 265–278.
[16] S. Burckhardt, D. Leijen, J. Protzenko, M. Fähndrich, Global Sequence Protocol: A Robust Abstraction for Replicated Shared State, ECOOP, vol. 37, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 568–590.
[17] D. Ongaro, J.K. Ousterhout, In search of an understandable consensus algorithm, in: 2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014, USENIX Association, pp. 305–319, 2014.
[18] H.C. Melgratti, C. Roldán, A formal analysis of the global sequence protocol, in: COORDINATION, vol. 9686, Springer, 2016, pp. 175–191.
[19] S.C. Stephens, From models to implementations - distributed algorithms using maude, Undergraduate thesis, University of Illinois at Urbana-Champaign, 2018.
[20] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155.
[21] P.C. Ölveczky (Ed.), Designing Reliable Distributed Systems, 2017.
[22] X. Zhao, OACP verification source code in maude repository, https://github.com/CynthiaZ92/OACP-Maude, 2019.
[23] Twissandra, Twitter clone on Cassandra, http://twissandra.com/, 2014. (Accessed 26 February 2018).
[24] P.A. Bernstein, S. Burckhardt, S. Bykov, N. Crooks, J.M. Faleiro, G. Kliot, A. Kumbhare, M.R. Rahman, V. Shah, A. Szekeres, J. Thelin, Geo-distribution of actor-based services, 1 (2017) 107:1–107:26.
[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of aspectj, in: Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, pp. 327–353.
[26] P. Hunt, M. Konar, F.P. Junqueira, B. Reed, Zookeeper: wait-free coordination for Internet-scale systems, in: USENIX Annual Technical Conference, USENIX Association, 2010.
[27] J.R. Goodman, Cache consistency and sequential consistency, Tech. rep., IEEE Scalable Coherence Interface Working Group, Mar. 1989.
[28] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, C. Hauser, Managing update conflicts in Bayou, aweakly connected replicated storage system, in: Proceedings 15th Symposium on Operating Systems Principles, 1995, pp. 172–183.
[29] S. Burckhardt, A. Gotsman, H. Yang, Understanding eventual consistency, Tech. rep., Microsoft Research, 2013.
[30] T. Kraska, M. Hentschel, G. Alonso, D. Kossmann, Consistency rationing in the cloud: pay only when it matters, PVLDB 2 (1) (2009) 253–264.
[31] J. Li, M.N. Krohn, D. Mazieres, D. Shasha, Secure untrusted data repository (SUNDR), in: Proc. 6th Symposium on Operating System Design and Implementation, USENIX Association, 2004, pp. 121–136.
[32] Shasha Mazieres, Building secure file systems out of byzantine storage, in: PODC: 21th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 2002.
[33] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N.M. Preguiça, M. Najafzadeh, M. Shapiro, Putting consistency back into eventual consistency, in: EuroSys, ACM, 2015, pp. 6:1–6:16.
[34] R. Ladin, B. Liskov, S. Ghemawat, Providing high availability using lazy replication, ACM Trans. Comput. Syst. 10 (4) (1992) 360–391.

[35] D.D. Akkoorath, J. Brandão, A. Bieniusa, C. Baquero, Global-local view: scalable consistency for concurrent data types, in: Euro-Par, vol. 11014, Springer, 2018, pp. 492–504.
[36] C. Li, J. Leitão, A. Clement, N.M. Preguiça, R. Rodrigues, V. Vafeiadis, Automating the choice of consistency levels in replicated systems, in: USENIX Annual Technical Conference, USENIX Association, 2014, pp. 281–292.
[37] P. Haller, On the integration of the actor model in mainstream technologies: the Scala perspective, in: AGERE!@SPLASH, 2012, pp. 1–6.
[38] G.A. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems, The MIT Press, Cambridge, MA, USA, 1986.
[39] R. Guerraoui, M. Pavlovic, D.-A. Seredinschi, Incremental consistency guarantees for replicated objects, arXiv:1609.02434 [abs], 2016.
[40] K.C. Sivaramakrishnan, G. Kaki, S. Jagannathan, Declarative programming over eventually consistent data stores, in: PLDI, ACM, 2015, pp. 413–424.
[41] P. Bernstein, S. Bykov, A. Geller, G. Kliot, J. Thelin, Orleans: distributed virtual actors for programmability and scalability, Tech. rep., March 2014.
[42] L. Lamport, The part-time Parliament, ACM Trans. Comput. Syst. 16 (2) (1998) 133–169.
[43] F.P. Junqueira, B.C. Reed, M. Serafini, Zab: high-performance broadcast for primary-backup systems, in: DSN, IEEE Compute Society, 2011, pp. 245–256.
[44] S. Burckhardt, M. Fähndrich, D. Leijen, B.P. Wood, Cloud types for eventual consistency, in: ECOOP, vol. 7313, Springer, 2012, pp. 283–307.
[45] P.A. Bernstein, S. Bykov, Developing cloud services using the orleans virtual actor model, 20 (2016) 71–75.
[46] P. Alvaro, P. Bailis, N. Conway, J.M. Hellerstein, Consistency without borders, in: ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013, 2013, pp. 23:1–23:10.
[47] C. Meiklejohn, P.V. Roy, Lasp: a language for distributed, coordination-free programming, in: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, ACM, 2015, pp. 184–195.
[48] F. Houshmand, M. Lesani, Hamsaz: replication coordination analysis and synthesis, PACMPL 3 (POPL) (2019) 74:1–74:32.
[49] A. Lakshman, P. Malik, Cassandra: a decentralized structured storage system, 44 (2010) 35–40.
[50] W. Lloyd, M.J. Freedman, M. Kaminsky, D.G. Andersen, Don't settle for eventual: scalable causal consistency for wide-area storage with COPS, in: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, ACM Press, 2011, pp. 401–416.
[51] W. Lloyd, M.J. Freedman, M. Kaminsky, D.G. Andersen, Stronger semantics for low-latency geo-replicated storage, in: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, Lombard, IL, USA, April 2-5, 2013, USENIX Association, 2013, pp. 313–328.
[52] S. Almeida, J. Leitão, L.E.T. Rodrigues, ChainReaction: a causal+ consistent datastore based on chain replication, in: EuroSys, ACM, 2013, pp. 85–98.
[53] G. Kaki, K. Sivaramakrishnan, S. Abeysiriwardane, S. Jagannathan, Mergeable types, in: ML Workshop, 2017.