Replicated Data Types: Specification, Verification, Optimality

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, Marek Zawirski

Replicated Data Types: Specification, Verification, Optimality

Sebastian Burckhardt
Microsoft Research

Alexey Gotsman
IMDEA Software Institute

Hongseok Yang
University of Oxford

Marek Zawirski
INRIA & UPMC-LIP6

Abstract

Geographically distributed systems often rely on replicated eventually consistent data stores to achieve availability and performance. To resolve conflicting updates at different replicas, researchers and practitioners have proposed specialized consistency protocols, called replicated data types, that implement objects such as registers, counters, sets or lists. Reasoning about replicated data types has however not been on par with comparable work on abstract data types and concurrent data types, lacking specifications, correctness proofs, and optimality results.

To fill in this gap, we propose a framework for specifying replicated data types using relations over events and verifying their implementations using replication-aware simulations. We apply it to 7 existing implementations of 4 data types with nontrivial conflict-resolution strategies and optimizations (last-writer-wins register, counter, multi-value register and observed-remove set). We also present a novel technique for obtaining lower bounds on the worst-case space overhead of data type implementations and use it to prove optimality of 4 implementations. Finally, we show how to specify consistency of replicated stores with multiple objects axiomatically, in analogy to prior work on weak memory models. Overall, our work provides foundational reasoning tools to support research on replicated eventually consistent stores.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Replication; eventual consistency; weak memory

1. Introduction

To achieve availability and scalability, many networked computing systems rely on *replicated stores*, allowing multiple clients to issue operations on shared data on a number of *replicas*, which communicate changes to each other using message passing. For example, large-scale Internet services rely on *geo-replication*, which places data replicas in geographically distinct locations, and applications for mobile devices store replicas locally to support offline use. One benefit of such architectures is that the replicas remain locally available to clients even when network connections fail. Unfortunately, the famous CAP theorem [19] shows that such high **A**vailability and tolerance to network **P**artitions are incompatible with *strong* **C***onsistency*, i.e., the illusion of a single centralized replica handling all operations. For this reason, modern replicated stores often provide weaker forms of consistency, commonly dubbed *eventual consistency* [36]. 'Eventual' usually refers to the guarantee that

> if clients stop issuing update requests, then the replicas will eventually reach a consistent state. (1)

Eventual consistency is a hot research area, and new replicated stores implementing it appear every year [1, 13, 16, 18, 23, 27, 33, 34, 37]. Unfortunately, their semantics is poorly understood: the very term eventual consistency is a catch-all buzzword, and different stores claiming to be eventually consistent actually provide subtly different guarantees. The property (1), which is a form of *quiescent consistency*, is too weak to capture these. Although it requires the replicas to converge to the same state eventually, it doesn't say which one it will be. Furthermore, (1) does not provide any guarantees in realistic scenarios when updates never stop arriving. The difficulty of reasoning about the behavior of eventually consistent stores comes from a multitude of choices to be made in their design, some of which we now explain.

Allowing the replicas to be temporarily inconsistent enables eventually consistent stores to satisfy clients' requests from the local replica immediately, and broadcast the changes to the other replicas only after the fact, when the network connection permits this. However, this means that clients can concurrently issue conflicting operations on the same data item at different replicas; furthermore, if the replicas are out-of-sync, these operations will be applied to its copies in different states. For example, two users sharing an online store account can write two different zip codes into the delivery address; the same users connected to replicas with different views of the shopping cart can also add and concurrently remove the same product. In such situations the store needs to ensure that, after the replicas exchange updates, the changes by different clients will be merged and all conflicts will be resolved in a meaningful way. Furthermore, to ensure eventual consistency (1), the conflict resolution has to be uniform across replicas, so that, in the end, they converge to the same state.

The protocols achieving this are commonly encapsulated within *replicated data types* [1, 10, 16, 18, 31, 33, 34] that implement *objects*, such as registers, counters, sets or lists, with various conflict-resolution strategies. The strategies can be as simple as establishing a total order on all operations using timestamps and letting the last writer win, but can also be much more subtle. Thus, a data type can detect the presence of a conflict and let the client deal with it: e.g., the *multi-value register* used in Amazon's Dynamo key-value store [18] would return both conflicting zip codes in the above example. A data type can also resolve the conflict in an application-specific way. For example, the *observed-remove set* [7, 32] processes concurrent operations trying to add and remove the same element so that an add always wins, an outcome that may be appropriate for a shopping cart.

Replicated data type implementations are often nontrivial, since they have to maintain not only client-observable object state, but also *metadata* needed to detect and resolve conflicts and to handle network failures. This makes reasoning about their behavior challenging. The situation gets only worse if we consider multi-

POPL '14, January 22–24, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2535838.2535848

ple replicated objects: in this case, asynchronous propagation of updates between replicas may lead to counterintuitive behaviors—*anomalies*, in database terminology. The following code illustrates an anomaly happening in real replicated stores [1, 18]:

$$\begin{array}{l}\text{Replica } r_1 \to x.\texttt{wr}(\textit{post}) \\ \qquad\quad y.\texttt{wr}(\textit{comment})\end{array} \;\Big\|\; \begin{array}{l} i = y.\texttt{rd}\ \textit{// comment} \leftarrow \text{Replica } r_2 \\ j = x.\texttt{rd}\ \textit{// empty}\end{array} \tag{2}$$

We have two clients reading from and writing to register objects x and y at two different replicas; i and j are client-local variables. The first client makes a post by writing to x at replica r_1 and then comments on the post by writing to y. After every write, replica r_1 might send a message with the update to replica r_2. If the messages carrying the writes of *post* to x and *comment* to y arrive to replica r_2 out of the order they were issued in, the second client can see the comment, but not the post. Different replicated stores may allow such an anomaly or not, and this has to be taken into account when reasoning about them.

In this paper, we propose techniques for reasoning about eventually consistent replicated stores in the following three areas.

1. Specification. We propose a comprehensive framework for specifying the semantics of replicated stores. Its key novel component is *replicated data type specifications* (§3), which provide the first way of specifying the semantics of replicated objects with advanced conflict resolution declaratively, like abstract data types [25]. We achieve this by defining the result of a data type operation not by a function of states, but of *operation contexts*—sets of events affecting the result of the operation, together with some relationships between them. We show that our specifications are sufficiently flexible to handle data types representing a variety of conflict-resolution strategies: last-write-wins register, counter, multi-value register and observed-remove set.

We then specify the semantics of a whole store with multiple objects, possibly of different types, by *consistency axioms* (§7), which constrain the way the store processes incoming requests in the style of weak shared-memory models [2] and thus define the anomalies allowed. As an illustration, we define consistency models used in existing replicated stores, including a weak form of eventual consistency [1, 18] and different kinds of causal consistency [23, 27, 33, 34]. We find that, when specialized to last-writer-wins registers, these specifications are very close to fragments of the C/C++ memory model [5]. Thus, our specification framework generalizes axiomatic shared-memory models to replicated stores with nontrivial conflict resolution.

2. Verification. We propose a method for proving the correctness of replicated data type implementations with respect to our specifications and apply it to seven existing implementations of the four data types mentioned above, including those with nontrivial optimizations. Reasoning about the implementations is difficult due to the highly concurrent nature of a replicated store, with multiple replicas simultaneously updating their object copies and exchanging messages. We address this challenge by proposing *replication-aware simulations* (§5). Like classical simulations from data refinement [21], these associate a concrete state of an implementation with its abstract description—structures on events, in our case. To combat the complexity of replication, they consider the state of an object at a single replica or a message in transit separately and associate it with abstract descriptions of only those events that led to it. Verifying an implementation then requires only reasoning about an instance of its code running at a single replica.

Here, however, we have to deal with another challenge: code at a single replica can access both the state of an object and a message at the same time, e.g., when updating the former upon receiving the latter. To reason about such code, we often need to rely on certain *agreement properties* correlating the abstract descriptions of the message and the object state. Establishing these properties requires global reasoning. Fortunately, we find that agreement properties needed to prove realistic implementations depend only on basic facts about their messaging behavior and can thus be established once for broad classes of data types. Then a particular implementation within such a class can be verified by reasoning purely locally.

By carefully structuring reasoning in this way, we achieve easy and intuitive proofs of single data type implementations. We then lift these results to stores with multiple objects of different types by showing how consistency axioms can be proved given properties of the transport layer and data type implementations (§7).

3. Optimality. Replicated data type designers strive to optimize their implementations; knowing that one is optimal can help guide such efforts in the most promising direction. However, proving optimality is challengingly broad as it requires quantifying over all possible implementations satisfying the same specification.

For most data types we studied, the primary optimization target is the size of the metadata needed to resolve conflicts or handle network failures. To establish optimality of metadata size, we present a novel method for proving lower bounds on the *worst-case metadata overhead* of replicated data types—the proportion of metadata relative to the client-observable content. The main idea is to find a large family of executions of an arbitrary correct implementation such that, given the results of data type operations from a certain fixed point in any of the executions, we can recover the previous execution history. This implies that, across executions, the states at this point are distinct and thus must have some minimal size.

Using our method, we prove that four of the implementations we verified have an optimal worst-case metadata overhead among all implementations satisfying the same specification. Two of these (counter, last-writer-wins register) are well-known; one (optimized observed-remove set [6]) is a recently proposed nontrivial optimization; and one (optimized multi-value register) is a small improvement of a known implementation [33] that we discovered during a failed attempt to prove optimality of the latter. We summarize all the bounds we proved in Fig. 10.

We hope that the theoretical foundations we develop will help in exploring the design space of replicated data types and replicated eventually consistent stores in a systematic way.

2. Replicated Data Types

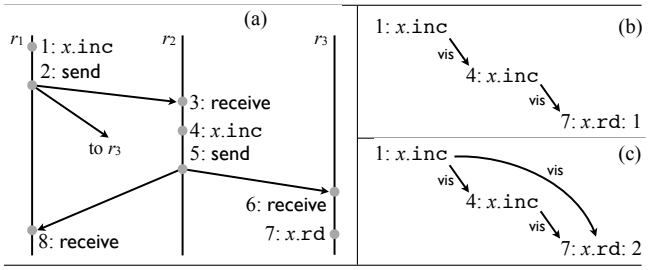
We now describe our formal model for replicated stores and introduce *replicated data type implementations*, which implement operations on a single object at a replica and the protocol used by replicas to exchange updates to this object. Our formalism follows closely the models used by replicated data type designers [33].

A replicated store is organized as a collection of named ***objects*** $\mathsf{Obj} = \{x, y, z, \ldots\}$. Each object is hosted at all ***replicas*** $r, s \in \mathsf{ReplicaID}$. The sets of objects and replicas may be infinite, to model their dynamic creation. Clients interact with the store by performing ***operations*** on objects at a specified replica. Each object $x \in \mathsf{Obj}$ has a ***type*** $\tau = \mathsf{type}(x) \in \mathsf{Type}$, whose ***type signature*** $(\mathsf{Op}_\tau, \mathsf{Val}_\tau)$ determines the set of supported operations Op_τ (ranged over by o) and the set of their return values Val_τ (ranged over by a, b, c, d). We assume that a special value $\bot \in \mathsf{Val}_\tau$ belongs to all sets Val_τ and is used for operations that return no value. For example, we can define a counter data type `ctr` and an integer register type `intreg` with operations for reading, incrementing or writing an integer a: $\mathsf{Val}_{\texttt{ctr}} = \mathsf{Val}_{\texttt{intreg}} = \mathbb{Z} \cup \{\bot\}$, $\mathsf{Op}_{\texttt{ctr}} = \{\texttt{rd}, \texttt{inc}\}$ and $\mathsf{Op}_{\texttt{intreg}} = \{\texttt{rd}\} \cup \{\texttt{wr}(a) \mid a \in \mathbb{Z}\}$.

We also assume sets $\mathsf{Message}$ of messages (ranged over by m) and timestamps $\mathsf{Timestamp}$ (ranged over by t). For simplicity, we let timestamps be positive integers: $\mathsf{Timestamp} = \mathbb{N}_1$.

DEFINITION 1. *A **replicated data type implementation** for a data type τ is a tuple $\mathcal{D}_\tau = (\Sigma, \vec{\sigma}_0, M, \mathsf{do}, \mathsf{send}, \mathsf{receive})$, where $\vec{\sigma}_0$:*

Figure 1. Illustrations of a concrete (a) and two abstract executions (b, c)



$\mathsf{ReplicaID} \to \Sigma$, $M \subseteq \mathsf{Message}$ *and*

$$\mathsf{do} : \mathsf{Op}_\tau \times \Sigma \times \mathsf{Timestamp} \to \Sigma \times \mathsf{Val}_\tau;$$
$$\mathsf{send} : \Sigma \to \Sigma \times M; \qquad \mathsf{receive} : \Sigma \times M \to \Sigma.$$

We denote a component of $\mathcal{D}_\tau$, such as do, by $\mathcal{D}_\tau.\mathsf{do}$. A tuple $\mathcal{D}_\tau$ defines the class of implementations of objects with type τ, meant to be instantiated for every such object in the store. Σ is the set of states (ranged over by σ) used to represent the current state of the object, including metadata, at a single replica. The initial state at every replica is given by $\vec{\sigma}_0$.

$\mathcal{D}_\tau$ provides three methods that the rest of the store implementation can call at a given replica; we assume that these methods execute atomically. We visualize store executions resulting from repeated calls to the methods as in Fig. 1(a), by arranging the calls on several vertical timelines corresponding to replicas at which they occur and denoting the delivery of messages by diagonal arrows. In §4, we formalize them as sequences of transitions called *concrete executions* and define the store semantics by their sets; the intuition given by Fig. 1(a) should suffice for the following discussion.

A client request to perform an operation $o \in \mathsf{Op}_\tau$ triggers the call $\mathsf{do}(o, \sigma, t)$ (e.g., event 1 in Fig. 1(a)). This takes the current state $\sigma \in \Sigma$ of the object at the replica where the request is issued and a timestamp $t \in \mathsf{Timestamp}$ provided by the rest of the store implementation and produces the updated object state and the return value of the operation. The data type implementation can use the timestamp provided, e.g., to implement the last-writer-wins conflict-resolution strategy mentioned in §1, but is free to ignore it.

Nondeterministically, in moments when the network is able to accept messages, a replica calls send. Given the current state of the object at the replica, send produces a message in M to broadcast to all other replicas (event 2 in Fig. 1(a)); sometimes send also alters the state of the object. Using broadcast rather than point-to-point communication does not limit generality, since we can always tag messages with the intended receiver. Another replica that receives the message generated by send calls receive to merge the enclosed update into its copy of the object state (event 3 in Fig. 1(a)).

We now reproduce three replicated data type implementations due to Shapiro et al. [33].They fall into two categories: in ***op-based*** implementations, each message carries a description of the latest operations that the sender has performed, and in ***state-based*** implementations, a description of *all* operations it knows about.

Op-based counter (`ctr`). Fig. 2(a) shows an implementation of the `ctr` data type. A replica stores a pair $\langle a, d\rangle$, where a is the current value of the counter, and d is the number of increments performed since the last broadcast (we use angle brackets for tuples representing states and messages). The send method returns d and resets it; the receive method adds the content of the message to a. This implementation is correct, as long as each message is delivered exactly once (we show how to prove this in §5). Since `inc` operations commute, they never conflict: applying them in different orders at different replicas yields the same final state.

State-based counter (`ctr`). The implementation in Fig. 2(b) summarizes the currently known history by recording the contribution of every replica to the counter value separately (reminiscent of vector clocks [29]). A replica stores its identifier r and a vector v such that for each replica s the entry $v(s)$ gives the number of increments made by clients at s that have been received by r. A `rd` operation returns the sum of all entries in the vector. An `inc` operation increments the entry for the current replica. We denote by $v[i \mapsto j]$ the function that has the same value as v everywhere, except for i, where it has the value j. The send method returns the vector, and the receive method takes the maximum of each entry in the vectors v and v' given to it. This is correct because an entry for s in either vector reflects a prefix of the sequence of increments done at replica s. Hence, we know that $\min\{v(s), v'(s)\}$ increments by s are taken into account both in $v(s)$ and in $v'(s)$.

Figure 2. Three replicated data type implementations

(a) Op-based counter (`ctr`)

Σ	$= \mathbb{N}_0 \times \mathbb{N}_0$	$\mathsf{do}(\mathtt{rd}, \langle a, d\rangle, t)$	$= (\langle a, d\rangle, a)$
M	$= \mathbb{N}_0$	$\mathsf{do}(\mathtt{inc}, \langle a, d\rangle, t)$	$= (\langle a+1, d+1\rangle, \bot)$
$\vec{\sigma}_0$	$= \lambda r.\, \langle 0, 0\rangle$	$\mathsf{send}(\langle a, d\rangle)$	$= (\langle a, 0\rangle, d)$
		$\mathsf{receive}(\langle a, d\rangle, d')$	$= \langle a + d', d\rangle$

(b) State-based counter (`ctr`)

Σ	$=$	$\mathsf{ReplicaID} \times (\mathsf{ReplicaID} \to \mathbb{N}_0)$
σ_0	$=$	$\lambda r.\, \langle r, \lambda s.\, 0\rangle$
M	$=$	$\mathsf{ReplicaID} \to \mathbb{N}_0$
$\mathsf{do}(\mathtt{rd}, \langle r, v\rangle, t)$	$=$	$(\langle r, v\rangle, \sum\{v(s) \mid s \in \mathsf{ReplicaID}\})$
$\mathsf{do}(\mathtt{inc}, \langle r, v\rangle, t)$	$=$	$(\langle r, v[r \mapsto v(r)+1]\rangle, \bot)$
$\mathsf{send}(\langle r, v\rangle)$	$=$	$(\langle r, v\rangle, v)$
$\mathsf{receive}(\langle r, v\rangle, v')$	$=$	$\langle r, (\lambda s.\, \max\{v(s), v'(s)\})\rangle$

(c) State-based last-writer-wins register (`intreg`)

Σ	$=$	$\mathbb{Z} \times (\mathsf{Timestamp} \uplus \{0\})$
$\vec{\sigma}_0$	$=$	$\lambda r.\, \langle 0, 0\rangle$
M	$=$	Σ
$\mathsf{do}(\mathtt{rd}, \langle a, t\rangle, t')$	$=$	$(\langle a, t\rangle, a)$
$\mathsf{do}(\mathtt{wr}(a'), \langle a, t\rangle, t')$	$=$	if $t < t'$ then $(\langle a', t'\rangle, \bot)$ else $(\langle a, t\rangle, \bot)$
$\mathsf{send}(\langle a, t\rangle)$	$=$	$(\langle a, t\rangle, \langle a, t\rangle)$
$\mathsf{receive}(\langle a, t\rangle, \langle a', t'\rangle)$	$=$	if $t < t'$ then $\langle a', t'\rangle$ else $\langle a, t\rangle$

State-based last-writer-wins (LWW) register (`intreg`). Unlike counters, registers have update operations that are not commutative. To resolve conflicts, the implementation in Fig. 2 uses the last-writer-wins strategy, creating a total order on writes by associating a unique timestamp with each of them. A state contains the current value, returned by `rd`, and the timestamp at which it was written (initially, we have 0 instead of a timestamp). A $\mathtt{wr}(a')$ compares its timestamp t' with the timestamp t of the current value a and sets the value to the one with the highest timestamp. Note that here we have to allow for $t' < t$, since we do not make any assumptions about timestamps apart from uniqueness: e.g., the rest of the store implementation can compute them using physical or Lamport clocks [22]. We show how to state assumptions about timestamps in §4. The send method just returns the state, and the receive method chooses the winning value by comparing the timestamps in the current state and the message, like `wr`.

State-based vs. op-based. State-based implementations converge to a consistent state faster than op-based implementations because they are ***transitively delivering***, meaning that they can propagate updates indirectly. For example, when using the counter in Fig. 2(b), in the execution in Fig. 1(a) the read at r_3 (event 7) returns 2, even though the message from r_1 has not arrived yet, because r_3 learns about r_1's update via r_2. State-based implementations are also resilient against transport failures like message *loss, reordering, or duplication*. Op-based implementations require the replicated store using them to mask such failures (e.g., using message sequence numbers, retransmission buffers, or reorder buffers).

The potential weakness of state-based implementations is the size of states and messages, which motivates our examination of space optimality in §6. For example, we show that the counter in Fig. 2(b) is optimal, meaning that no counter implementation satisfying the same requirements (transitive delivery and resilience against message loss, reordering, and duplication) can do better.

3. Specifying Replicated Data Types and Stores

Consider the concrete execution in Fig. 1(a). What are valid return values for the read in event 7? Intuitively, 1 or 2 can be justifiable, but not 100. We now present a framework for specifying the expected outcome declaratively, without referring to implementation details. For example, we give a specification of a replicated counter that is satisfied by both implementations in Fig. 2(a, b).

In presenting the framework, we rely on the intuitive understanding of the way a replicated store executes given in §2. Later we define the store semantics formally (§4), which lets us state what it means for a store to satisfy our specifications (§4 and §7).

3.1 Abstract Executions and Specification Structure

We define our specifications on *abstract executions*, which include only user-visible events (corresponding to do calls) and describe the other information about the store processing in an implementation-independent form. Informally, we consider a concrete execution correct if it can be justified by an abstract execution satisfying the specifications that is "similar" to it and, in particular, has the same operations and return values.

Abstract executions are inspired by axiomatic definitions of weak shared-memory models [2]. In particular, we use their previously proposed reformulation with visibility and arbitration relations [13], which are similar to the reads-from and coherence relations from weak shared-memory models. We provide a comparison with shared-memory models in §7 and with [13] in §8.

DEFINITION 2. *An **abstract execution** is a tuple* $A = (E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar})$*, where*

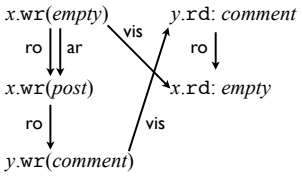
- $E \subseteq \mathsf{Event}$ *is a set of events from a countable universe* Event*;*
- *each event* $e \in E$ *describes a replica* $\mathsf{repl}(e) \in \mathsf{ReplicaID}$ *performing an operation* $\mathsf{oper}(e) \in \mathsf{Op}_{\mathsf{type}(\mathsf{obj}(e))}$ *on an object* $\mathsf{obj}(e) \in \mathsf{Obj}$*, which returns the value* $\mathsf{rval}(e) \in \mathsf{Val}_{\mathsf{type}(\mathsf{obj}(e))}$*;*
- $\mathsf{ro} \subseteq E \times E$ *is a **replica order**, which is a union of transitive, irreflexive and total orders on events at each replica;*
- $\mathsf{vis} \subseteq E \times E$ *is an acyclic **visibility relation** such that* $\forall e, f \in E.\, e \xrightarrow{\mathsf{vis}} f \implies \mathsf{obj}(e) = \mathsf{obj}(f)$*;*
- $\mathsf{ar} \subseteq E \times E$ *is an **arbitration relation**, which is a union of transitive, irreflexive and total orders on events on each object.*

We also require that ro*,* vis *and* ar *be well-founded.*

In the following, we denote components of A and similar structures as in $A.\mathsf{repl}$. We also use $(e, f) \in \mathsf{r}$ and $e \xrightarrow{\mathsf{r}} f$ interchangeably.

Informally, $e \xrightarrow{\mathsf{vis}} f$ means that f is aware of e and thus e's effect can influence f's return value. In implementation terms, this may be the case if the update performed by e has been delivered to the replica performing f before f is issued. The exact meaning of "delivered", however, depends on how much information messages carry in the implementation. For example, as we explain in §3.2, the return value of a read from a counter is equal to the number of `inc` operations visible to it. Then, as we formalize in §4, the abstract execution illustrated in Fig. 1(b) justifies the op-based implementation in Fig. 2(a) reading 1 in the concrete execution in Fig. 1(a). The abstract execution in Fig. 1(c) justifies the state-based implementation in Fig. 2(b) reading 2 due to transitive delivery (§2). There is no abstract execution that would justify reading 100.

The ar relation represents the ordering information provided by the store, e.g., via timestamps. On the right we show an abstract execution corresponding to a variant of the anomaly (2). The ar edge means that any replica that sees both writes to x should assume that *post* overwrites *empty*.



We give a store specification by two components, constraining abstract executions:

1. ***Replicated data type specifications*** determine return values of operations in an abstract execution in terms of its vis and ar relations, and thus define conflict-resolution policies for individual objects in the store. The specifications are the key novel component of our framework, and we discuss them next.
2. ***Consistency axioms*** constrain vis and ar and thereby disallow anomalies and extend the semantics of individual objects to that of the entire store. We defer their discussion to §7. See Fig. 13 for their flavor; in particular, COCV prohibits the anomaly above.

Each of these components can be varied separately, and our specifications will define the semantics of any possible combination. Given a specification of a store, we can determine whether a set of events can be observed by its users by checking if there is an abstract execution with this set of events satisfying the data type specifications and consistency axioms.

3.2 Replicated Data Type Specifications

In a sequential setting, the semantics of a data type τ can be specified by a function $\mathcal{S}_\tau : \mathsf{Op}_\tau^+ \to \mathsf{Val}_\tau$, which, given a non-empty sequence of operations performed on an object, specifies the return value of the last operation. For a register, read operations return the value of the last preceding write, or zero if there is no prior write. For a counter, read operations return the number of preceding increments. Thus, for any sequence of operations ξ:

$$\begin{aligned} \mathcal{S}_{\mathtt{intreg}}(\xi\,\mathtt{rd}) &= a, \text{ if } \mathtt{wr}(0)\,\xi = \xi_1\,\mathtt{wr}(a)\,\xi_2 \text{ and} \\ &\qquad \xi_2 \text{ does not contain } \mathtt{wr} \text{ operations;} \\ \mathcal{S}_{\mathtt{ctr}}(\xi\,\mathtt{rd}) &= \text{(the number of } \mathtt{inc} \text{ operations in } \xi\text{);} \\ \mathcal{S}_{\mathtt{intreg}}(\xi\,\mathtt{inc}) &= \mathcal{S}_{\mathtt{ctr}}(\xi\,\mathtt{wr}(a)) = \bot. \end{aligned}$$

In a replicated store, the story is more interesting. We specify a data type τ by a function $\mathcal{F}_\tau$, generalizing $\mathcal{S}_\tau$. Just like $\mathcal{S}_\tau$, this determines the return value of an operation based on prior operations performed on the object. However, $\mathcal{F}_\tau$ takes as a parameter not a sequence, but an *operation context*, which includes all we need to know about a store execution to determine the return value of a given operation o—the set E of all events that are visible to o, together with the operations performed by the events and visibility and arbitration relations on them.

DEFINITION 3. *An **operation context** for a data type τ is a tuple $L = (o, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar})$, where $o \in \mathsf{Op}_\tau$, E is a finite subset of* Event*, $\mathsf{oper} : E \to \mathsf{Op}_\tau$, $\mathsf{vis} \subseteq E \times E$ is acyclic and $\mathsf{ar} \subseteq E \times E$ is transitive, irreflexive and total.*

We can extract the context of an event $e \in A.E$ in an abstract execution A by selecting all events visible to it according to $A.\mathsf{vis}$:

$$\mathsf{ctxt}(A, e) = (A.\mathsf{oper}(e), G, (A.\mathsf{oper})|_G, (A.\mathsf{vis})|_G, (A.\mathsf{ar})|_G),$$

where $G = (A.\mathsf{vis})^{-1}(e)$ and $\cdot|_G$ is the restriction to events in G. Thus, in the abstract execution in Fig. 1(b), the operation context of the read from x includes only one increment event; in the execution in Fig. 1(c) it includes two.

DEFINITION 4. *A **replicated data type specification** for a type τ is a function $\mathcal{F}_\tau$ that, given an operation context L for τ, specifies a return value $\mathcal{F}_\tau(L) \in \mathsf{Val}_\tau$.*

Note that $\mathcal{F}_\tau(o, \emptyset, \ldots)$ returns the value resulting from performing o on the initial state for the data type (e.g., 0 for the LWW-register).

We specify multiple data types used in a replicated store by a partial function $\mathbb{F}$ mapping them to data type specifications.

DEFINITION 5. *An abstract execution A **satisfies** $\mathbb{F}$, written $A \models \mathbb{F}$, if the return value of every event in A is computed on its context by the specification for the type of the object the event accesses:*

$$\forall e \in A.E.\ (A.\mathsf{rval}(e) = \mathbb{F}(\mathsf{type}(A.\mathsf{obj}(e)))(\mathsf{ctxt}(A, e))).$$

We specify a whole store by $\mathbb{F}$ and a set of consistency axioms (§7). This lets us determine if its users can observe a given set of events by checking if there is an abstract execution with these events that satisfies $\mathbb{F}$ according to the above definition, as well as the axioms.

Note that $\mathcal{F}_\tau$ is deterministic. This does not mean that so is an outcome of an operation on a store; rather, that all the nondeterminism arising due to its distributed nature is resolved by vis and ar in the context passed to $\mathcal{F}_\tau$. These relations are chosen arbitrarily subject to consistency axioms. Due to the determinacy property, two events that perform the same operation and see the same set of events produce the same return values. As we show in §7, this property ensures that our specifications can formalize eventual consistency in the sense of (1).

We now give four examples of data type specifications, corresponding to the four conflict-resolution strategies mentioned in §1 and §2: (1) operations commute, so no conflicts arise; (2) last writer wins; (3) all conflicting values are returned; and (4) conflicts are resolved in an application-specific way. We start by specifying the data types whose implementations we presented in §2.

1. Counter (`ctr`) is defined by

$$\begin{aligned} \mathcal{F}_{\mathtt{ctr}}(\mathtt{inc}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) &= \bot; \\ \mathcal{F}_{\mathtt{ctr}}(\mathtt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) &= \left|\{e \in E \mid \mathsf{oper}(e) = \mathtt{inc}\}\right|. \end{aligned} \quad (3)$$

Thus, according to Def. 5 the executions in Fig. 1(b) and 1(c) satisfy the counter specification: both 1 and 2 are valid return values for the read from x when there are two concurrent increments.

2. LWW-register (`intreg`) is defined by

$$\mathcal{F}_{\mathtt{intreg}}(o, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \mathcal{S}_{\mathtt{intreg}}(E^{\mathsf{ar}} o), \quad (4)$$

where E^{ar} denotes the sequence obtained by ordering the operations performed by the events in E according to ar. Thus, the return value is determined by establishing a total order of the visible operations and applying the regular sequential semantics. For example, by Def. 5 in the example execution from §3.1 the read from x has to return *empty*; if we had a vis edge from the write of *post* to the read from x, then the read would have to return *post*. As we show in §7, weak shared-memory models are obtained by specializing our framework to stores with only LWW-registers.

We can obtain a concurrent semantics $\mathcal{F}_\tau$ of any data type τ based on its sequential semantics $\mathcal{S}_\tau$ similarly to (4). For example, $\mathcal{F}_{\mathtt{ctr}}$ defined above is equivalent to what we obtain using this generic construction. The next two examples go beyond this.

3. Multi-value register (`mvr`). This register [1, 18] has the same operations as the LWW-register, but its reads return a set of values:

$$\begin{aligned} \mathcal{F}_{\mathtt{mvr}}(\mathtt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E.\, \mathsf{oper}(e) = \mathtt{wr}(a) \\ \wedge \neg\exists f \in E.\, \mathsf{oper}(f) = \mathtt{wr}(\_) \wedge e \xrightarrow{\mathsf{vis}} f\}. \end{aligned}$$

(We write _ for an expression whose value is irrelevant.) A read returns the values written by currently conflicting writes, defined as those that are not superseded in vis by later writes; ar is not used. For example, a `rd` would return $\{2, 3\}$ in the context on the right.

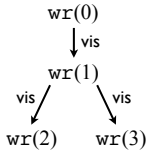


Figure 3. The set of configurations Config and the transition relation $\longrightarrow_{\mathbb{D}}$: Config × Event × Config for a data type library $\mathbb{D}$. We use $e : \{h_1 = u_1, h_2 = u_2\}$ to abbreviate $h_1(e) = u_1$ and $h_2(e) = u_2$. We uncurry $R \in \mathsf{RState}$ where convenient.

$$\begin{aligned} \mathsf{Obj}_\tau &= \{x \in \mathsf{Obj} \mid \mathsf{type}(x) = \tau\} \\ \mathsf{RState} &= \textstyle\bigcup_{X \subseteq \mathsf{Obj}} \prod_{x \in X} (\mathsf{ReplicaID} \to \mathbb{D}(\mathsf{type}(x)).\Sigma) \\ \mathsf{TState} &= \mathsf{MessageID} \rightharpoonup \textstyle\bigcup_{\tau \in \mathsf{Type}} (\mathsf{ReplicaID} \times \mathsf{Obj}_\tau \times \mathbb{D}(\tau).M) \\ \mathsf{Config} &= \mathsf{RState} \times \mathsf{TState} \end{aligned}$$

$$\frac{\mathbb{D}(\mathsf{type}(x)).\mathsf{do}(o, \sigma, t) = (\sigma', a) \quad e : \{\mathsf{act} = \mathsf{do},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{oper} = o,\ \mathsf{time} = t,\ \mathsf{rval} = a\}}{(R[(x,r) \mapsto \sigma],\ T) \xrightarrow{e}_{\mathbb{D}} (R[(x,r) \mapsto \sigma'],\ T)}$$

$$\frac{\mathbb{D}(\mathsf{type}(x)).\mathsf{send}(\sigma) = (\sigma', m) \qquad mid \notin \mathsf{dom}(T) \quad e : \{\mathsf{act} = \mathsf{send},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{msg} = mid\}}{(R[(x,r) \mapsto \sigma],\ T) \xrightarrow{e}_{\mathbb{D}} (R[(x,r) \mapsto \sigma'],\ T[mid \mapsto (r, x, m)])}$$

$$\frac{\mathbb{D}(\mathsf{type}(x)).\mathsf{receive}(\sigma, m) = \sigma' \qquad r \neq r' \quad e : \{\mathsf{act} = \mathsf{receive},\ \mathsf{obj} = x,\ \mathsf{repl} = r,\ \mathsf{srepl} = r',\ \mathsf{msg} = mid\}}{(R[(x,r) \mapsto \sigma],\ T[mid \mapsto (r', x, m)]) \xrightarrow{e}_{\mathbb{D}} (R[(x,r) \mapsto \sigma'],\ T[mid \mapsto (r', x, m)])}$$

4. Observed-remove set (`orset`). How do we specify a replicated set of integers? The operations of adding and removing different elements commute and thus do not conflict. Conflicts arise from concurrently adding and removing the same element. For example, we need to decide what `rd` will return as the contents of the set in the context $(\mathtt{rd}, \{e, f\}, \mathsf{oper}, \mathsf{vis}, \mathsf{ar})$, where $\mathsf{oper}(e) = \mathtt{add}(42)$ and $\mathsf{oper}(f) = \mathtt{remove}(42)$. If we use the generic construction from the LWW-register, the result will depend on the arbitration relation: $\emptyset$ if $e \xrightarrow{\mathsf{ar}} f$, and $\{42\}$ otherwise. An application may require a more consistent behavior, e.g., that an add operation always win against concurrent remove operations. Observed-remove (OR) set [7, 32] achieves this by mandating that `remove` operations cancel only the `add` operations that are visible to them:

$$\begin{aligned} \mathcal{F}_{\mathtt{orset}}(\mathtt{rd}, E, \mathsf{oper}, \mathsf{vis}, \mathsf{ar}) = \{a \mid \exists e \in E.\, \mathsf{oper}(e) = \mathtt{add}(a) \\ \wedge \neg\exists f \in E.\, \mathsf{oper}(f) = \mathtt{remove}(a) \wedge e \xrightarrow{\mathsf{vis}} f\}, \end{aligned} \quad (5)$$

In the above operation context `rd` will return $\emptyset$ if $e \xrightarrow{\mathsf{vis}} f$, and $\{42\}$ otherwise. The rationale is that, in the former case, $\mathtt{add}(42)$ and $\mathtt{remove}(42)$ are not concurrent: the user who issued the `remove` knew that 42 was in the set and thus meant to remove it. In the latter case, the two operations are concurrent and thus `add` wins.

As the above examples illustrate, our specifications can describe the semantics of data types and their conflict-resolution policies declaratively, without referring to the internals of their implementations. In this sense the specifications generalize the concept of an *abstract data type* [25] to the replicated setting.

4. Store Semantics and Data Type Correctness

A ***data type library*** $\mathbb{D}$ is a partial mapping from types τ to data type implementations $\mathbb{D}(\tau)$ from Def. 1. We now define the semantics of a replicated store with a data type library $\mathbb{D}$ as a set of its *concrete executions*, previously introduced informally by Fig. 1(a). We then state what it means for data type implementations of §2 to satisfy their specifications of §3.2 by requiring their concrete executions to be justified by abstract ones. In §7 we generalize this to the correctness of the whole store with multiple object with respect to both data type specifications and consistency axioms.

Semantics. We define the semantics using the relation $\longrightarrow_{\mathbb{D}}$: Config × Event × Config in Fig. 3, which describes a single

step of the store execution. The relation transforms ***configurations*** $(R, T) \in \mathsf{Config}$ describing the store state: R gives the object state at each replica, and T the set of messages in transit between them, each identified by a message identifier $mid \in \mathsf{MessageID}$. A message is annotated by the origin replica and the object to which it pertains. We allow the store to contain only some objects from Obj and thus allow R to be partial on them. We use a number of functions on events, such as act, obj, etc., to record the information about the corresponding transitions, so that $\longrightarrow_{\mathbb{D}}$ is implicitly parameterized by them; we give their full list in Def. 6 below.

The first rule in Fig. 3 describes a replica r performing an operation o on an object x using the do method of the corresponding data type implementation. We record the return value using the function rval. To communicate the change to other replicas, we can at any time perform a transition defined by the second rule, which puts a new message m created by a call to send into the set of messages in transit. The third rule describes the delivery of such a message to a replica r other than the origin replica r', which triggers a call to $\mathsf{receive}$. Note that the relation $\longrightarrow_{\mathbb{D}}$ does not make any assumptions about message delivery: messages can be delivered in any order, multiple times, or not at all. These assumptions can be introduced separately, as we show later in this section. A concrete execution can be thought of as a finite or infinite sequence of transitions:

$$(R_0, T_0) \xrightarrow{e_1} (R_1, T_1) \xrightarrow{e_2} \dots \xrightarrow{e_n} (R_n, T_n) \dots,$$

where all events e_i are distinct. To ease mapping between concrete and abstract executions in the future, we formalize it as a structure on events, similarly to Def. 2.

DEFINITION 6. *A **concrete execution** of a store with a data type library $\mathbb{D}$ is a tuple*

$$C = (E, \mathsf{eo}, \mathsf{pre}, \mathsf{post}, \mathsf{act}, \mathsf{obj}, \mathsf{repl}, \mathsf{oper}, \mathsf{time}, \mathsf{rval}, \mathsf{msg}, \mathsf{srepl}).$$

*Here $E \subseteq \mathsf{Event}$, the **execution order** eo is a well-founded, transitive, irreflexive and total order on E, relating the events according to the order of the transitions they describe, time is injective and $\mathsf{pre}, \mathsf{post} : E \to \mathsf{Config}$ form a valid sequence of transitions:*

$$(\forall e \in E.\, \mathsf{pre}(e) \xrightarrow{e}_{\mathbb{D}} \mathsf{post}(e)) \wedge$$
$$(\forall e, f \in E.\, e \xrightarrow{\mathsf{eo}} f \wedge \neg\exists g.\, e \xrightarrow{\mathsf{eo}} g \xrightarrow{\mathsf{eo}} f \implies \mathsf{post}(e) = \mathsf{pre}(f)).$$

We have omitted the types of functions on events, which are easily inferred from Fig. 3: e.g., $\mathsf{act} : E \to \{\mathsf{do}, \mathsf{send}, \mathsf{receive}\}$ and $\mathsf{time} : E \rightharpoonup \mathsf{Timestamp}$, defined only on e with $\mathsf{act}(e) = \mathsf{do}$.

We denote the initial configuration of C by $\mathsf{init}(C) = C.\mathsf{pre}(e_0)$, where e_0 is the minimal event in $C.\mathsf{eo}$. If $C.E$ is finite, we denote the final configuration of C by $\mathsf{final}(C) = C.\mathsf{post}(e_\mathfrak{f})$, where $e_\mathfrak{f}$ is the maximal event in $C.\mathsf{eo}$. The ***semantics*** $[\![\mathbb{D}]\!]$ of $\mathbb{D}$ is the set of all its concrete executions C that start in a configuration with an empty set of messages and all objects in initial states, i.e.,

$$\exists X \subseteq \mathsf{Obj}.\, \mathsf{init}(C) = ((\lambda x \in X.\, \mathbb{D}(\mathsf{type}(x)).\vec{\sigma}_0), [\,]),$$

where $[\,]$ is the everywhere-undefined function.

Transport layer specifications. Data type implementations such as the op-based counter in Fig. 2(a) can rely on some guarantees concerning the delivery of messages ensured by the rest of the store implementation. They may similarly assume certain properties of timestamps other than uniqueness (guaranteed by the injectivity of time in Def. 6). We take such assumptions into account by admitting only a subset of executions from $[\![\mathbb{D}]\!]$ that satisfy a ***transport layer specification*** $\mathcal{T}$, which is a predicate on concrete executions. Thus, we consider a ***replicated store*** to be defined by a pair $(\mathbb{D}, \mathcal{T})$ and the set of its executions be $[\![\mathbb{D}]\!] \cap \mathcal{T}$.

Even though our definition of $\mathcal{T}$ lets it potentially restrict data type implementation internals, the particular instantiations we use only restrict message delivery and timestamps. For technical reasons, we assume that $\mathcal{T}$ always satisfies certain closure properties: for every $C \in \mathcal{T}$, the projection of C onto events on a given object or a subset of events forming a prefix in the eo order is also in $\mathcal{T}$.

As an example, we define a transport layer specification ensuring that a message is delivered to any single replica at most once, as required by the implementation in Fig. 2(a). Let the ***delivery relation*** $\mathsf{del}(C) \in C.E \times C.E$ pair events sending and receiving the same message:

$$e \xrightarrow{\mathsf{del}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \wedge C.\mathsf{act}(e) = \mathsf{send} \wedge$$
$$C.\mathsf{act}(f) = \mathsf{receive} \wedge C.\mathsf{msg}(e) = C.\mathsf{msg}(f).$$

Then the desired condition on concrete executions C is

$$\forall e, f, g \in C.E.\, e \xrightarrow{\mathsf{del}(C)} f \wedge e \xrightarrow{\mathsf{del}(C)} g \wedge$$
$$C.\mathsf{repl}(f) = C.\mathsf{repl}(g) \implies f = g. \quad (\mathsf{T\text{-}Unique})$$

Data type implementation correctness. We now state what it means for an implementation $\mathcal{D}_\tau$ of type τ from Def. 1 to satisfy a specification $\mathcal{F}_\tau$ from Def. 4. To this end, we consider the behavior of $\mathcal{D}_\tau$ under the "most general" client and transport layer, performing all possible operations and message deliveries. Formally, let $[\![\mathcal{D}_\tau]\!]$ be the set of executions $C \in [\![[\tau \mapsto \mathcal{D}_\tau]]\!]$ of a store containing a single object x of a type τ with the implementation $\mathcal{D}_\tau$, i.e., $\mathsf{init}(C) = (R, [\,])$ for some R such that $\mathsf{dom}(R) = \{x\}$.

Then $\mathcal{D}_\tau$ should satisfy $\mathcal{F}_\tau$ under a transport specification $\mathcal{T}$ if for every concrete execution $C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$ we can find a "similar" abstract execution satisfying $\mathcal{F}_\tau$ and, in particular, having the same operations and return values. As it happens, all components of the abstract execution except visibility are straightforwardly determined by C; as explained in §3.1, we have some freedom in choosing visibility. We define the choice using a ***visibility witness*** $\mathcal{V}$, which maps a concrete execution $C \in [\![\mathcal{D}_\tau]\!]$ to an acyclic relation on $(C.E)|_{\mathsf{do}}$ defining visibility (here $\cdot|_{\mathsf{do}}$ is the restriction to events e with $C.\mathsf{act}(e) = \mathsf{do}$). Let

$$e \xrightarrow{\mathsf{ro}(C)} f \iff e \xrightarrow{C.\mathsf{eo}} f \wedge C.\mathsf{repl}(e) = C.\mathsf{repl}(f);$$
$$e \xrightarrow{\mathsf{ar}(C)} f \iff e, f \in (C.E)|_{\mathsf{do}} \wedge$$
$$C.\mathsf{obj}(e) = C.\mathsf{obj}(f) \wedge C.\mathsf{time}(e) < C.\mathsf{time}(f).$$

Then the abstract execution justifying $C \in [\![\mathcal{D}_\tau]\!]$ is defined by

$$\mathsf{abs}(C, \mathcal{V}) = (C.E|_{\mathsf{do}}, E.\mathsf{repl}|_{\mathsf{do}}, E.\mathsf{obj}|_{\mathsf{do}}, E.\mathsf{oper}|_{\mathsf{do}}, E.\mathsf{rval}|_{\mathsf{do}},$$
$$\mathsf{ro}(C)|_{\mathsf{do}}, \mathcal{V}(C), \mathsf{ar}(C)).$$

DEFINITION 7. *A data type implementation $\mathcal{D}_\tau$ **satisfies** a specification $\mathcal{F}_\tau$ with respect to $\mathcal{V}$ and $\mathcal{T}$, written $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$, if $\forall C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}.\, (\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau])$, where $\models$ is defined in Def. 5.*

As we explained informally in §3.1, the visibility witness depends on how much information the implementation puts into messages. Since state-based implementations, such as the ones in Fig. 2(b, c), are transitively delivering (§2), for them we use the witness $\mathcal{V}^{\mathsf{state}}(C) = (\mathsf{ro}(C) \cup \mathsf{del}(C))^+|_{\mathsf{do}}$. By the definition of $\mathsf{ro}(C)$ and $\mathsf{del}(C)$, $(\mathsf{ro}(C) \cup \mathsf{del}(C))$ is acyclic, so $\mathcal{V}^{\mathsf{state}}$ is well-defined. State-based implementations do not make any assumptions about the transport layer: in this case we write $\mathcal{T} = \mathsf{T\text{-}Any}$. In contrast, op-based implementations, such as the one in Fig. 2(a), require $\mathcal{T} = \mathsf{T\text{-}Unique}$. Since such implementations are not transitively delivering, the witness $\mathcal{V}^{\mathsf{state}}$ is not appropriate for them. We could attempt to define a witness for them by straightforwardly lifting the delivery relation:

$$\mathcal{V}^{\mathsf{op}}(C) = \mathsf{ro}(C)|_{\mathsf{do}} \cup \{(e, f) \mid e, f \in (C.E)|_{\mathsf{do}} \wedge$$
$$\exists e', f'.\, e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f\}.$$

However, we need to be more careful, since for op-based implementations $e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f$ does not ensure that the update of e is taken into account by f: if there is another send event e'' in between e and e', then e'' will capture the update of e and e' will not. Hence, we define the witness as:

$$\begin{aligned}\mathcal{V}^{\mathsf{op}}(C) = \mathsf{ro}(C)|_{\mathsf{do}} \cup \{(e,f) \mid e,f \in (C.E)|_{\mathsf{do}} \\ \wedge\, \exists e', f'.\, e \xrightarrow{\mathsf{ro}(C)} e' \xrightarrow{\mathsf{del}(C)} f' \xrightarrow{\mathsf{ro}(C)} f \\ \wedge\, \neg\exists e''.\, e \xrightarrow{\mathsf{ro}(C)} e'' \xrightarrow{\mathsf{ro}(C)} e' \wedge C.\mathsf{act}(e'') = \mathsf{send}\}.\end{aligned}$$

We next present a method for proving data type implementation correctness in the sense of Def. 7. In §7 we lift this to stores with multiple objects and take into account consistency axioms.

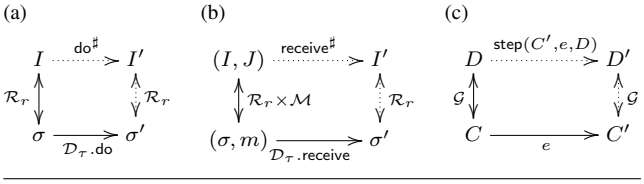
5. Proving Data Type Implementations Correct

The straightforward approach to proving correctness in the sense of Def. 7 would require us to consider global store configurations in executions C, including object states at all replicas and all messages in transit, making the reasoning non-modular and unintuitive. To deal with this challenge, we focus on a single component of a store configuration using *replication-aware simulation relations* $\mathcal{R}_r$ and $\mathcal{M}$, analogous to simulation (aka coupling) relations used in data refinement [21]. The $\mathcal{R}_r$ relation associates the object state at a replica r with an abstract execution that describes *only those events that led to this state*; $\mathcal{M}$ does the same for a message. For example, when proving $\mathcal{D}_{\mathtt{ctr}}$ in Fig. 2(b) with respect to $\mathcal{F}_{\mathtt{ctr}}$ in (3), $\mathcal{M}$ associates a message carrying a vector v with executions in which each replica s makes $v(s)$ increments. As part of a proof of $\mathcal{D}_\tau$, we require checking that the effect of its methods, such as $\mathcal{D}_\tau$.do, can be simulated by appropriately transforming related abstract executions while preserving the relations. We define these transformations using *abstract methods* $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$ and $\mathsf{receive}^\sharp$ as illustrated in Fig. 4(a, b). For example, if a replica r executes $\mathcal{D}_\tau$.do from a state σ related by $\mathcal{R}_r$ to an abstract execution I (we explain the use of I instead of A later), we need to find an I' related by $\mathcal{R}_r$ to the resulting state σ'. We also need to check that the value returned by $\mathcal{D}_\tau$.do on σ is equal to that returned by $\mathcal{F}_\tau$ on I.

These conditions consider the behavior of an implementation method on a single state and/or message and its effect on only the relevant part of the abstract execution. However, by localizing the reasoning in this way, we lose some global information that is actually required to verify realistic implementations. In particular, this occurs when discharging the obligation for receive in Fig. 4(b). Taking a global view, σ and m there are meant to come from the same configuration in a concrete execution C; correspondingly, I and J are meant to be fragments of the same abstract execution $\mathsf{abs}(C, \mathcal{V})$. In this context we may know certain ***agreement properties*** correlating I and J, e.g., that the union of their visibility relations is itself a well-formed visibility relation and is thus acyclic. Establishing them requires global reasoning about whole executions C and $\mathsf{abs}(C, \mathcal{V})$. Fortunately, we find that this can be done knowing only the abstract methods, not the implementation $\mathcal{D}_\tau$. Furthermore, these methods state basic facts about the messaging behavior of implementations and are thus common to broad classes of them, such as state-based or op-based. This allows us to establish agreement properties using global reasoning once for a given class of implementations; at this stage we can also benefit from the transport layer specification $\mathcal{T}$ and check that the abstract methods construct visibility according to the given witness $\mathcal{V}$. Then a particular implementation within the class can be verified by discharging local obligations, such as those in Fig. 4(a, b), while assuming agreement properties. This yields easy and intuitive proofs.

To summarize, we deal with the challenge posed by a distributed data type implementation by decomposing reasoning about it into global reasoning done once for a broad class of implementations and local implementation-specific reasoning. We start by presenting the general form of obligations to be discharged for a single implementation within a certain class (§5.1) and the particular form they take for the class of state-based implementations (§5.2), together with some examples (§5.3). We then formulate the obligations to be discharged for a class of implementations (§5.4), which in particular, establish the agreement properties assumed in the per-implementation obligations. In [12, §B], we give the obligations for op-based implementations, together with a proof of the counter in Fig. 2(a). An impatient reader can move on to §6 after finishing §5.3, and come back to §5.4 later.

Figure 4. Diagrams illustrating replication-aware simulations



Since Def. 7 considers only single-object executions, we fix an object x of type τ and consider only concrete and abstract executions over x, whose sets we denote by $\mathsf{CEx}[x]$ and $\mathsf{AEx}[x]$.

5.1 Replication-Aware Simulations

As is typical for simulation-based proofs, we need to use auxiliary state to record information about the computation history. For this reason, actually our simulation relations associate a state or a message with an ***instrumented execution***—a pair $(A, \mathsf{info}) \in \mathsf{IEx}$ of an abstract execution $A \in \mathsf{AEx}[x]$ and a function $\mathsf{info} : A.E \to \mathsf{AInfo}$, tagging events with auxiliary information from a set AInfo. As we show below, AInfo can be chosen once for a class of data type implementations: e.g., $\mathsf{AInfo} = \mathsf{Timestamp}$ for state-based ones (§5.2). We use I and J to range over instrumented executions and shorten, e.g., $I.A.E$ to $I.E$. For a partial function h we write $h(x)\!\downarrow$ for $x \in \mathsf{dom}(h)$, and adopt the convention that $h(x) = y$ implies $h(x)\!\downarrow$.

DEFINITION 8. *A **replication-aware simulation** between $\mathcal{D}_\tau$ and $\mathcal{F}_\tau$ with respect to* info *and **abstract methods** $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$ and $\mathsf{receive}^\sharp$ is a collection of relations $\{\mathcal{R}_r, \mathcal{M} \mid r \in \mathsf{ReplicaID}\}$ satisfying the conditions in Fig. 5.*

Here info and abstract methods are meant to be fixed for a given class of implementations, such as state or op-based. To prove a particular implementation within this class, one needs to find simulation relations satisfying the conditions in Fig. 5. For example, as we show in §5.3, the following relation lets us prove the correctness of the counter in Fig. 2(b) with respect to info and abstract methods appropriate for state-based implementations:

$$\begin{aligned}&\langle s, v\rangle\ [\mathcal{R}_r]\ I \iff (r = s) \wedge (v\ [\mathcal{M}]\ I);\\ &v\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff \\ &\qquad \forall s.\, v(s) = \big|\{e \in E \mid \mathsf{oper}(e) = \mathtt{inc} \wedge \mathsf{repl}(e) = s\}\big|.\end{aligned} \tag{6}$$

INIT in Fig. 5 associates the initial state at a replica r with the execution having an empty set of events. DO, SEND and RECEIVE formalize the obligations illustrated in Fig. 4(a, b). Note that $\mathsf{do}^\sharp$ is parameterized by an event e (required to be fresh in instantiations) and the information about the operation performed.

The abstract methods are partial and the obligations in Fig. 5 *assume* that their applications are defined. When instantiating $\mathsf{receive}^\sharp$ for a given class of implementations, we let it be defined only when its arguments satisfy the agreement property for this class, which we establish separately (§5.4). While doing this, we can also establish some ***execution invariants***, holding of single ex-

Figure 5. Definition of a replication-aware simulation $\{\mathcal{R}_r, \mathcal{M}\}$ between $\mathcal{D}_\tau$ and $\mathcal{F}_\tau$. All free variables in each condition are implicitly universally quantified and have the following types: $\sigma, \sigma' \in \Sigma$, $m \in M$, $I, I', J \in \mathsf{IEx}$, $e \in \mathsf{Event}$, $r \in \mathsf{ReplicaID}$, $o \in \mathsf{Op}_\tau$, $a \in \mathsf{Val}_\tau$, $t \in \mathsf{Timestamp}$.

$$\mathcal{R}_r \subseteq \Sigma \times \mathsf{IEx},\ r \in \mathsf{ReplicaID}; \qquad \mathcal{M} \subseteq M \times \mathsf{IEx}$$

$$\mathsf{do}^\sharp : \mathsf{IEx} \times \mathsf{Event} \times \mathsf{ReplicaID} \times \mathsf{Op}_\tau \times \mathsf{Val}_\tau \times \mathsf{Timestamp} \rightharpoonup \mathsf{IEx}$$

$$\mathsf{send}^\sharp : \mathsf{IEx} \rightharpoonup \mathsf{IEx} \times \mathsf{IEx}; \qquad \mathsf{receive}^\sharp : \mathsf{IEx} \times \mathsf{IEx} \rightharpoonup \mathsf{IEx}$$

INIT: $\mathcal{D}_\tau.\vec{\sigma}_0(r)\ [\mathcal{R}_r]\ I_\emptyset$, where $I_\emptyset.E = \emptyset$

DO: $(\mathsf{do}^\sharp(I, e, r, o, a, t) = I' \ \wedge\ \mathcal{D}_\tau.\mathsf{do}(o, \sigma, t) = (\sigma', a) \ \wedge\ \sigma\ [\mathcal{R}_r]\ I)$
$\implies (\sigma'\ [\mathcal{R}_r]\ I' \ \wedge\ a = \mathcal{F}_\tau(\mathsf{ctxt}(I'.A, e)))$

SEND: $(\mathsf{send}^\sharp(I) = (I', J) \ \wedge\ \mathcal{D}_\tau.\mathsf{send}(\sigma) = (\sigma', m) \ \wedge\ \sigma\ [\mathcal{R}_r]\ I)$
$\implies (\sigma'\ [\mathcal{R}_r]\ I' \ \wedge\ m\ [\mathcal{M}]\ J)$

RECEIVE: $(\mathsf{receive}^\sharp(I, J)\downarrow \ \wedge\ \sigma\ [\mathcal{R}_r]\ I \ \wedge\ m\ [\mathcal{M}]\ J)$
$\implies (\mathcal{D}_\tau.\mathsf{receive}(\sigma, m)\ [\mathcal{R}_r]\ \mathsf{receive}^\sharp(I, J))$

ecutions supplied as parameters to $\mathsf{do}^\sharp$ and $\mathsf{send}^\sharp$. We similarly assume them in Fig. 5 via the definedness of these abstract methods.

5.2 Instantiation for State-Based Implementations

Fig. 6 defines the domain AInfo and abstract methods appropriate for state-based implementations. In §5.4 we show that the existence of a simulation of Def. 8 with respect to these parameters implies $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_\tau$ (Theorems 9 and 10). The $\mathsf{do}^\sharp$ method adds a fresh event e with the given attributes to I; its timestamp t is recorded in info. In the resulting execution I', the event e is the last one by its replica, observes all events in I and occupies the place in arbitration consistent with t. The $\mathsf{send}^\sharp$ method just returns I, which formalizes the intuition that, in state-based implementations, send returns a message capturing all the information about the object available at the replica. The $\mathsf{receive}^\sharp$ method takes the component-wise union $I \sqcup J$ of executions I related to the current state and J related to the message, applied recursively to the components of $I.A$ and $J.A$. We also assume that $I \sqcup J$ recomputes the arbitration relation in the resulting execution from the timestamps. This is the reason for recording them in info: we would not be able to construct $\mathsf{receive}^\sharp(I, J).\mathsf{ar}$ solely from $I.\mathsf{ar}$ and $J.\mathsf{ar}$.

The agreement property $\mathsf{agree}(I, J)$ guarantees that $I \sqcup J$ is well-formed (e.g., its visibility relation is acyclic) and that, for each replica, I describes a computation extending J or vice versa. The latter follows from the observation we made when explaining the state-based counter in §2: a message or a state in a state-based implementation reflects a prefix of the sequence of events performed at a given replica. The first conjunct of the execution invariant inv requires arbitration to be consistent with event timestamps; the second conjunct follows from the definition of $\mathcal{V}^{\mathsf{state}}$ (§4). When discharging the obligations in Fig. 5 with respect to the parameters in Fig. 6 for a particular implementation, we can rely on the agreement property and the execution invariant.

5.3 Examples

We illustrate the use of the instantiation from §5.2 on the state-based counter, LWW-register and OR-set. In [12, §A] we also give proofs of two multi-value register implementations.

Counter: $\mathcal{D}_{\mathtt{ctr}}$ in Fig. 2(b) and $\mathcal{F}_{\mathtt{ctr}}$ in (3). Discharging the obligations in Fig. 5 for the simulation (6) is easy. The key case is RECEIVE, where the first conjunct of agree in Fig. 6 ensures that $\min\{v(s), v'(s)\}$ increments by a replica s are taken into account both in $v(s)$ and in $v'(s)$:

$$(\langle r, v\rangle\ [\mathcal{R}_r]\ I) \wedge (v'\ [\mathcal{M}]\ J) \implies (\forall s.\ \min\{v(s), v'(s)\} = |\{e \in I.E \cap J.E \mid I.\mathsf{oper}(e) = \mathtt{inc} \wedge I.\mathsf{repl}(e) = s\}|).$$

Figure 6. Instantiation for state-based data type implementations. In $\mathsf{do}^\sharp$ we omit the straightforward definition of ar' in terms of $I.\mathsf{info}$ and t.

$$\mathsf{AInfo} = \mathsf{Timestamp}$$

$$\mathsf{agree}(I, J) \iff \forall r.\ (\{e \in I.E \mid I.\mathsf{repl}(e) = r\}, I.\mathsf{ro}) \text{ is a prefix of } (\{e \in J.E \mid J.\mathsf{repl}(e) = r\}, J.\mathsf{ro}) \text{ or vice versa}) \wedge (I \sqcup J \in \mathsf{IEx})$$

$$\mathsf{inv}(I) \iff (\forall e, f \in I.E.\ (e, f) \in I.\mathsf{ar} \iff I.\mathsf{info}(e) < I.\mathsf{info}(f)) \wedge ((I.\mathsf{vis} \cup I.\mathsf{ro})^+ \subseteq I.\mathsf{vis})$$

$$\mathsf{do}^\sharp(I, e, r, o, a, t) = I', \quad \text{if } \mathsf{inv}(I) \wedge e \notin I.E \wedge I' \in \mathsf{IEx}$$
$$\text{where } I' = ((I.E \cup \{e\},\ I.\mathsf{repl}[e \mapsto r],\ I.\mathsf{obj}[e \mapsto x],\ I.\mathsf{oper}[e \mapsto o],\ I.\mathsf{rval}[e \mapsto a],\ I.\mathsf{ro} \cup \{(f, e) \mid f \in I.E \wedge I.\mathsf{repl}(f) = r\},\ I.\mathsf{vis} \cup \{(f, e) \mid f \in I.E\}, \mathsf{ar}'),\ I.\mathsf{info}[e \mapsto t])$$

$$\mathsf{send}^\sharp(I) = (I, I), \quad \text{if } \mathsf{inv}(I)$$

$$\mathsf{receive}^\sharp(I, J) = I \sqcup J, \quad \text{if } \mathsf{inv}(I) \wedge \mathsf{inv}(J) \wedge \mathsf{agree}(I, J)$$

This allows establishing $\mathsf{receive}(\langle r, v\rangle, v')\ [\mathcal{R}_r]\ (I \sqcup J)$, thus formalizing the informal justification of correctness we gave in §2.

LWW-register: $\mathcal{D}_{\mathtt{intreg}}$ in Fig. 2(c) and $\mathcal{F}_{\mathtt{intreg}}$ in (4). We associate a state or a message $\langle a, t\rangle$ with any execution that contains a $\mathtt{wr}(a)$ event with the timestamp t maximal among all other $\mathtt{wr}$ events (as per info). By inv in Fig. 6, this event is maximal in arbitration, which implies that $\mathtt{rd}$ returns the correct value; the other obligations are also discharged easily. Formally, $\forall r.\ \mathcal{R}_r = \mathcal{M}$ and

$$\langle a, t\rangle\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff (t = 0 \wedge a = 0 \wedge (\neg\exists e \in E.\ \mathsf{oper}(e) = \mathtt{wr}(\_))) \vee (t > 0 \wedge (\exists e \in E.\ \mathsf{oper}(e) = \mathtt{wr}(a) \wedge \mathsf{info}(e) = t) \wedge (\forall f \in E.\ \mathsf{oper}(f) = \mathtt{wr}(\_) \implies \mathsf{info}(f) \leq t)).$$

Optimized OR-set: $\mathcal{D}_{\mathtt{orset}}$ in Fig. 7 and $\mathcal{F}_{\mathtt{orset}}$ in (5). A problem with implementing a replicated set is that we often cannot discard the information about an element from a replica state after it has been removed: if another replica unaware of the removal sends us a snapshot of its state containing this element, the semantics of the set may require our receive to keep the element out of the set. As we prove in §6, for the OR-set keeping track of information about removed elements cannot be fully avoided, which makes its space-efficient implementation very challenging. Here we consider a recently-proposed OR-set implementation [6] that, as we show in §6, has an optimal space complexity. It improves on the original implementation [32], whose complexity was suboptimal (we have proved the correctness of the latter as well; see [12, §A]).

An additional challenge posed by the OR-set is that, according to $\mathcal{F}_{\mathtt{orset}}$, a $\mathtt{remove}$ operation may behave differently with respect to different events adding the same element to the set, depending on whether it sees them or not. This causes the implementation to treat internally each $\mathtt{add}$ operation as generating a unique *instance* of the *element* being added, further increasing the space required. To combat this, the implementation concisely summarizes information about instances. An instance is represented by a unique *instance identifier* that is generated when a replica performs an $\mathtt{add}$ and consists of the replica identifier and the number of $\mathtt{add}$s (of any elements) performed at the replica until then. In a state $\langle r, V, w\rangle$, the vector w determines the identifiers of all instances that the current replica r has ever observed: for any replica s, the replica r has seen $w(s)$ successive identifiers $(s, 1), (s, 2), \ldots, (s, w(s))$ generated at s. To generate a new identifier in $\mathsf{do}(\mathtt{add}(a'))$, the replica r increments $w(r)$. The connection between the vector w in a state or a message and $\mathtt{add}$ events $e_{s,k}$ in corresponding executions is formalized in lines 1-3 of the simulation relation, also shown in Fig. 7. In receive we take the pointwise maximum of the two vectors w and w'. Like for the counter, the first conjunct of agree implies that this preserves the clauses in lines 1-3.

Figure 7. Optimized OR-set implementation [6] and its simulation

$\Sigma = \mathsf{ReplicaID} \times ((\mathbb{Z} \times \mathsf{ReplicaID}) \to \mathbb{N}_0) \times (\mathsf{ReplicaID} \to \mathbb{N}_0)$

$\vec{\sigma}_0 = \lambda r.\, \langle r, (\lambda a, s.\, 0), (\lambda s.\, 0)\rangle$

$M = ((\mathbb{Z} \times \mathsf{ReplicaID}) \to \mathbb{N}_0) \times (\mathsf{ReplicaID} \to \mathbb{N}_0)$

$\mathsf{do}(\mathtt{add}(a'), \langle r, V, w\rangle, t) = (\langle r, (\lambda a, s.\, \text{if } a = a' \wedge s = r \text{ then } w(r) + 1 \text{ else } V(a, s)), w[r \mapsto w(r) + 1]\rangle, \bot)$

$\mathsf{do}(\mathtt{remove}(a'), \langle r, V, w\rangle, t) = (\langle r, (\lambda a, s.\, \text{if } a = a' \text{ then } 0 \text{ else } V(a, s)), w\rangle, \bot)$

$\mathsf{do}(\mathtt{rd}, \langle r, V, w\rangle, t) = (\langle r, V, w\rangle, \{a \mid \exists s.\, V(a, s) > 0\})$

$\mathsf{send}(\langle r, V, w\rangle) = (\langle r, V, w\rangle, \langle V, w\rangle)$

$\mathsf{receive}(\langle r, V, w\rangle, \langle V', w'\rangle) = \langle r, (\lambda a, s.\, \text{if } (V(a, s) = 0 \wedge w(s) \geq V'(a, s)) \vee (V'(a, s) = 0 \wedge w'(s) \geq V(a, s)) \text{ then } 0 \text{ else } \max\{V(a, s), V'(a, s)\}), (\lambda s.\, \max\{w(s), w'(s)\})\rangle$

$\langle s, V, w\rangle\ [\mathcal{R}_\tau]\ I \iff (r = s) \wedge (\langle V, w\rangle\ [\mathcal{M}]\ I)$

$\langle V, w\rangle\ [\mathcal{M}]\ ((E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar}), \mathsf{info}) \iff$

1: $\exists \text{ distinct } e_{s,k}.\, (\{e_{s,k} \mid s \in \mathsf{ReplicaID} \wedge 1 \leq k \leq w(s)\} =$

2: $\quad \{e \in E \mid \mathsf{oper}(e) = \mathtt{add}(\_)\}) \wedge$

3: $(\forall s, k, j.\, (\mathsf{repl}(e_{s,k}) = s) \wedge (e_{s,j} \xrightarrow{\mathsf{ro}} e_{s,k} \iff j < k)) \wedge$

4: $(\forall a, s.\, (V(a, s) \leq w(s)) \wedge (V(a, s) \neq 0 \implies$

5: $\quad (\mathsf{oper}(e_{s,V(a,s)}) = \mathtt{add}(a)) \wedge$

6: $\quad (\neg\exists k.\, V(a, s) < k \leq w(s) \wedge \mathsf{oper}(e_{s,k}) = \mathtt{add}(a)) \wedge$

7: $\quad (\neg\exists f \in E.\, \mathsf{oper}(f) = \mathtt{remove}(a) \wedge e_{s,V(a,s)} \xrightarrow{\mathsf{vis}} f))) \wedge$

8: $(\forall a, s, k.\, e_{s,k} \in E \wedge \mathsf{oper}(e_{s,k}) = \mathtt{add}(a) \implies$

9: $\quad (k \leq V(a, s) \vee \exists f \in E.\, \mathsf{oper}(f) = \mathtt{remove}(a) \wedge e_{s,k} \xrightarrow{\mathsf{vis}} f))$

The component w in $\langle r, V, w\rangle$ records identifiers of both of those instances that have been removed and those that are still in the set (are *active*). The component V serves to distinguish the latter. As it happens, we do not need to store all active instances of an element a: for every replica s, it is enough to keep the last active instance identifier generated by an $\mathtt{add}(a)$ at this replica. If $V(a, s) \neq 0$, this identifier is $(s, V(a, s))$; if $V(a, s) = 0$, all instances of a generated at s that the current replica knows about are inactive. The meaning of V is formalized in the simulation: each instance identifier given by V is covered by w (line 4) and, if $V(a, s) \neq 0$, then the event $e_{s,V(a,s)}$ performs $\mathtt{add}(a)$ (line 5), is the last $\mathtt{add}(a)$ by replica s (line 6) and has not been observed by a $\mathtt{remove}(a)$ (line 7). Finally, the $\mathtt{add}(a)$ events that are not seen by a $\mathtt{remove}(a)$ in the execution are either the events $e_{s,V(a,s)}$ or those superseded by them (lines 8-9). This ensures that returning all elements with an active instance in $\mathtt{rd}$ matches $\mathcal{F}_{\mathtt{orset}}$.

When a replica r performs $\mathsf{do}(\mathtt{add}(a'))$, we update $V(a', r)$ to correspond to the new instance identifier. Conversely, in $\mathsf{do}(\mathtt{remove}(a'))$, we clear all entries in $V(a')$, thereby deactivating all instances of a'. However, after this their identifiers are still recorded in w, and so we know that they have been previously removed. This allows us to address the problem with implementing receive we mentioned above: if we receive a message with an active instance $(s, V'(a, s))$ of an element a that is not in the set at our replica ($V(a, s) = 0$), but previously existed ($w(s) \geq V'(a, s)$), this means that the instance has been removed and should not be active in the resulting state (the entry for (a, s) should be 0). We also do the same check with the state and the message swapped.

As the above explanation shows, our simulation relations are useful not only for proving correctness of data type implementations, but also for explaining their designs. Discharging obligations in Fig. 5 requires some work for the OR-set; due to space constraints, we defer this to [12, §A].

Figure 8. Function step that mirrors the effect of an event $e \in C'.E$ from $C' \in \mathsf{CEx}[x]$ in $D \in \mathsf{DEx}$, defined when so is the abstract method used

$\mathsf{step}(C', e, D) = D[r \mapsto \mathsf{do}^\sharp(D(r), e, r, C'.\mathsf{oper}(e), C'.\mathsf{rval}(e), C'.\mathsf{time}(e))]$,
if $C'.\mathsf{act}(e) = \mathsf{do} \wedge C'.\mathsf{repl}(e) = r$

$\mathsf{step}(C', e, D) = D[r \mapsto I, C'.\mathsf{msg}(e) \mapsto J]$,
if $C'.\mathsf{act}(e) = \mathsf{send} \wedge C'.\mathsf{repl}(e) = r \wedge C'.\mathsf{msg}(e) \notin \mathsf{dom}(D) \wedge \mathsf{send}^\sharp(D(r)) = (I, J)$

$\mathsf{step}(C', e, D) = D[r \mapsto \mathsf{receive}^\sharp(D(r), D(C'.\mathsf{msg}(e)))]$,
if $C'.\mathsf{act}(e) = \mathsf{receive} \wedge C'.\mathsf{repl}(e) = r$

5.4 Soundness and Establishing Agreement Properties

We present conditions on AInfo and abstract methods ensuring the soundness of replication-aware simulations over them and, in particular, establishing the agreement property and execution invariants assumed via the definedness of abstract operations in Fig. 5.

THEOREM 9 (Soundness). *Assume* AInfo, $\mathsf{do}^\sharp$, $\mathsf{send}^\sharp$, $\mathsf{receive}^\sharp$, $\mathcal{V}$ *and* $\mathcal{T}$ *that satisfy the conditions in Fig. 9 for some* $\mathcal{G}$. *If there exists a replication-aware simulation between* $\mathcal{D}_\tau$ *and* $\mathcal{F}_\tau$ *with respect to these parameters, then* $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$.

Conditions in Fig. 9 require global reasoning, but can be discharged once for a class of data types. For example, they hold of the instantiation for state-based implementations from §5.2, as well as one for op-based implementations presented in [12, §B].

THEOREM 10. *There exists* $\mathcal{G}$ *such that, for all* $\mathcal{D}_\tau$, *the parameters in Fig. 6 satisfy the conditions in Fig. 9 with respect to this* $\mathcal{G}$, $\mathcal{V} = \mathcal{V}^{\mathsf{state}}$, $\mathcal{T} = \mathsf{T\text{-}Any}$.

The proofs of Theorems 9 and 10 are given in [12, §B]. To explain the conditions in Fig. 9, here we consider the proof strategy for Theorem 9. To establish $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}, \mathcal{T}]\ \mathcal{F}_\tau$, for any $C \in [\![\mathcal{D}_\tau]\!] \cap \mathcal{T}$ we need to show $\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau]$. We prove this by induction on the length of C. To use the localized conditions in Fig. 5, we require a relation $\mathcal{G}$ associating C with a ***decomposed execution***—a partial function $D : (\mathsf{ReplicaID} \cup \mathsf{MessageID}) \rightharpoonup \mathsf{IEx}$ that gives fragments of $\mathsf{abs}(C, \mathcal{V})$ corresponding to replica states and messages in the final configuration of C. We write DEx for the set of all decomposed executions, so that $\mathcal{G} \subseteq \mathsf{CEx}[x] \times \mathsf{DEx}$. The existence of a decomposed execution D such that $C\ [\mathcal{G}]\ D$ forms the core of our induction hypothesis. G-CTXT in Fig. 9 checks that the abstract methods construct visibility according to $\mathcal{V}$: it requires the context of any event e by a replica r to be the same in $D(r)$ and $\mathsf{abs}(C, \mathcal{V})$. Together with DO in Fig. 5, this ensures $\mathsf{abs}(C, \mathcal{V}) \models [\tau \mapsto \mathcal{F}_\tau]$.

We write $C' \sim (C \xrightarrow{e} (R, T))$ when C' is an extension of C in the following sense: $C'.E = C.E \uplus \{e\}$, the other components of C are those of C' restricted to $C.E$, e is last in $C'.\mathsf{eo}$ and $C'.\mathsf{post}(e) = (R, T)$. For the induction step, assume $C\ [\mathcal{G}]\ D$ and $C' \sim (C \xrightarrow{e} (R, T))$; see Fig. 4(c). Then the decomposed execution D' corresponding to C' is given by $\mathsf{step}(C', e, D)$, where the function step in Fig. 8 mirrors the effect of the event e from C' in D using the abstract methods. G-STEP ensures that it preserves the relation $\mathcal{G}$. Crucially, G-STEP also requires us to establish the definedness of step and thus the corresponding abstract method. This justifies the agreement property and execution invariants encoded by the definedness and allows us to use the conditions in Fig. 5 to complete the induction. We also require G-INIT, which establishes the base case, and G-VIS, which formulates a technical restriction on $\mathcal{V}$. Finally, the conditions in Fig. 9 allow us to use the transport specification $\mathcal{T}$ by considering only executions C satisfying it.

6. Space Bounds and Implementation Optimality

Object states in replicated data type implementations include not only the current client-observable content, but also metadata

Figure 9. Proof obligations for abstract methods. Free variables are implicitly universally quantified and have the following types: $C, C' \in \mathsf{CEx}[x] \cap \mathcal{T}$, $D \in \mathsf{DEx}$, $r \in \mathsf{ReplicaID}$, $e \in \mathsf{Event}$, $(R, T) \in \mathsf{Config}$.

G-CTXT: $(C\ [\mathcal{G}]\ D \wedge e \in \mathsf{abs}(C, \mathcal{V}).E \wedge \mathsf{abs}(C, \mathcal{V}).\mathsf{repl}(e) = r) \implies \mathsf{ctxt}(D(r).A, e) = \mathsf{ctxt}(\mathsf{abs}(C, \mathcal{V}), e)$

G-STEP: $(C' \sim (C \xrightarrow{e} (R, T)) \wedge (C\ [\mathcal{G}]\ D)) \implies (\mathsf{step}(C', e, D)\downarrow \wedge C'\ [\mathcal{G}]\ \mathsf{step}(C', e, D))$

G-INIT: $(C.E = \{e\} \wedge C.\mathsf{pre}(e) = (\_, [\,])) \implies (\mathsf{step}(C, e, D_\emptyset)\downarrow \wedge C\ [\mathcal{G}]\ \mathsf{step}(C, e, D_\emptyset))$, where $D_\emptyset$ is such that $\mathsf{dom}(D_\emptyset) = \mathsf{ReplicaID} \wedge \forall r \in \mathsf{ReplicaID}.\ D_\emptyset(r).E = \emptyset$

G-VIS: $(e \in \mathsf{abs}(C, \mathcal{V}).E \wedge (C \text{ is a prefix of } C' \text{ under } C'.\mathsf{eo})) \implies \mathsf{ctxt}(\mathsf{abs}(C, \mathcal{V}), e) = \mathsf{ctxt}(\mathsf{abs}(C', \mathcal{V}), e)$

needed for conflict resolution or masking network failures. Space taken by this metadata is a major factor determining their efficiency and feasibility. As illustrated by the OR-set in §5.3, this is especially so for state-based implementations, i.e., those that satisfy their data type specifications with respect to the visibility witness $\mathcal{V}^{\mathsf{state}}$ and the transport layer specification $\mathsf{T\text{-}Any}$. We now present a general technique for proving lower bounds on this space overhead, which we use to prove optimality of four state-based implementations (we leave other implementation classes for future work; see §9). As in §5, we only consider executions over a fixed object x of type τ.

6.1 Metadata Overhead

To measure space, we need to consider how data are represented. An ***encoding*** of a set S is an injective function $\mathsf{enc} : S \to \Lambda^+$, where Λ is some suitably chosen fixed finite set of characters (left unspecified). Sometimes, we clarify the domain being encoded using a subscript: e.g., $\mathsf{enc}_{\mathbb{N}_0}(1)$. For $s \in S$, we let $\mathsf{len}_S(s)$ be the length of $\mathsf{enc}_S(s)$. The length can vary: e.g., for an integer k, $\mathsf{len}_{\mathbb{N}_0}(k) \in \Theta(\lg k)$. We use standard encodings (listed in [12, §C]) for return values $\mathsf{enc}_{\mathsf{Val}_\tau}$ of the data types τ we consider and assume an arbitrary but fixed encoding of object states $\mathsf{enc}_{\mathcal{D}_\tau.\Sigma}$.

To distinguish metadata from the client-observable content of the object, we assume that each data type has a special `rd` operation that returns the latter, as is the case in the examples considered so far. For a concrete execution $C \in [\![\mathcal{D}_\tau]\!]$ over the object x and a read event $e \in (C.E)|_{\mathtt{rd}}$, we define $\mathsf{state}(e)$ to be the state of the object accessed at e: $\mathsf{state}(e) = R(x, C.\mathsf{repl}(e))$ for $(R, \_) = C.\mathsf{pre}(e)$.

We now define the metadata overhead as a ratio, by dividing the size of the object state by the size of the observable state. We then quantify the worst-case overhead by taking the maximum of this ratio over all read operations in all executions with given numbers of replicas n and update operations m. To define the latter, we assume that each data type τ specifies a set $\mathsf{Upd}_\tau \subset \mathsf{Op}_\tau$ of update operations; for all examples in this paper $\mathsf{Upd}_\tau = \mathsf{Op}_\tau \setminus \{\mathtt{rd}\}$.

DEFINITION 11. *The **maximum metadata overhead** of an execution $C \in [\![\mathcal{D}_\tau]\!]$ of an implementation $\mathcal{D}_\tau$ is*

$$\mathsf{mmo}(\mathcal{D}_\tau, C) = \max\left\{ \frac{\mathsf{len}_{\mathcal{D}_\tau.\Sigma}(\mathsf{state}(e))}{\mathsf{len}_{\mathsf{Val}_\tau}(C.\mathsf{rval}(e))} \mid e \in (C.E)|_{\mathtt{rd}} \right\}.$$

*The **worst-case metadata overhead** of an implementation $\mathcal{D}_\tau$ over all executions with n replicas and m updates ($2 \le n \le m$) is*

$$\begin{aligned}\mathsf{wcmo}(\mathcal{D}_\tau, n, m) = \max\{\mathsf{mmo}(\mathcal{D}_\tau, C) \mid C \in [\![\mathcal{D}_\tau]\!] \wedge \\ n = |\{C.\mathsf{repl}(e) \mid e \in C.E\}| \wedge \\ m = |\{e \in C.E \mid C.\mathsf{oper}(e) \in \mathsf{Upd}_\tau\}|\}.\end{aligned}$$

We consider only executions with $m \ge n$, since we are interested in the asymptotic overhead of executions where all replicas are mutated (i.e., perform at least one update operation).

Figure 10. Summary of bounds on metadata overhead for stated-based implementations, as functions of the number of replicas n and updates m

Type	Existing implementation			Any implementation
	algorithm	ref.	overhead	overhead
`ctr`	Fig. 2(b)	[32]	$\widehat{\Theta}(n)$	$\widehat{\Omega}(n)$
`orset`	Fig. 7	[6]	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
	Fig. 15, [12, §A]	[32]	$\widehat{\Theta}(m \lg m)$	
`intreg`	Fig. 2(c)	[32]	$\widehat{\Theta}(\lg m)^\dagger$	$\widehat{\Omega}(\lg m)$
`mvr`	Fig. 17, [12, §A]	new‡	$\widehat{\Theta}(n \lg m)$	$\widehat{\Omega}(n \lg m)$
	Fig. 16, [12, §A]	[32]	$\widehat{\Theta}(n^2 \lg m)$	

† Assuming timestamp encoding is $O(\lg m)$, satisfied by Lamport clocks.
‡ An optimization of [32] discovered during the optimality proof.

DEFINITION 12. *Assume $\mathcal{D}_\tau$ and a positive function $f(n, m)$.*

- *f is an **asymptotic upper bound** ($\mathcal{D}_\tau \in \widehat{O}(f(n, m))$) if $\sup_{n,m\to\infty}(\mathsf{wcmo}(\mathcal{D}_\tau, n, m)/f(n, m)) < \infty$, i.e.,*

$$\exists K > 0.\, \forall m \ge n \ge 2.\, \mathsf{wcmo}(\mathcal{D}_\tau, n, m) < K f(n, m);$$

- *f is an **asymptotic lower bound** ($\mathcal{D}_\tau \in \widehat{\Omega}(f(n, m))$) if $\lim_{n,m\to\infty}(\mathsf{wcmo}(\mathcal{D}_\tau, n, m)/f(n, m)) \ne 0$, i.e.,*

$$\exists K > 0.\, \forall m_0 \ge n_0 \ge 2.\, \exists n \ge n_0, m \ge n_0.\ \mathsf{wcmo}(\mathcal{D}_\tau, n, m) > K f(n, m);$$

- *f is an **asymptotically tight bound** ($\mathcal{D}_\tau \in \widehat{\Theta}(f(n, m))$) if it is both an upper and a lower asymptotic bound.*

Fig. 10 summarizes our results; as described in §5, we have proved all the implementations correct. Matching lower and upper bounds indicate worst-case optimality of an implementation (note that this is different from optimality in all cases). The derivation of upper bounds relies on standard techniques and is deferred to [12, §C]. We now proceed to the main challenge: how to derive lower bounds that apply to *any* implementation of τ. We present proofs for `ctr` and `orset`; `intreg` and `mvr` are covered in [12, §C].

6.2 Experiment Families

The goal is to show that for any correct implementation $\mathcal{D}_\tau$ (i.e., such that $\mathcal{D}_\tau\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_\tau$), the object state must store some minimum amount of information. We achieve this by constructing an *experiment family*, which is a collection of executions $\mathbb{C}_\alpha$, where $\alpha \in Q$ for some index set Q. Each experiment contains a distinguished read event $\mathbf{e}_\alpha$. The experiments are designed in such a way that the object states $\mathsf{state}(\mathbf{e}_\alpha)$ must be distinct, which then implies a lower bound $\lg_{|\Lambda|} |Q|$ on the size of their encoding. To prove that they are distinct, we construct black-box tests that execute the methods of $\mathcal{D}_\tau$ on the states and show that the tests must produce different results for each $\mathsf{state}(\mathbf{e}_\alpha)$ provided $\mathcal{D}_\tau$ is correct. Formally, the tests induce a read-back function rb that satisfies $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. We encapsulate the core argument in the following lemma.

DEFINITION 13. *An **experiment family** for an implementation $\mathcal{D}_\tau$ is a tuple $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ where Q is a finite set, $2 \le n \le m$, and for each $\alpha \in Q$, $\mathbb{C}_\alpha \in [\![\mathcal{D}_\tau]\!]$ is an execution with n replicas and m updates, $\mathbf{e}_\alpha \in (\mathbb{C}_\alpha.E)|_{\mathtt{rd}}$ and $\mathsf{rb} : \mathcal{D}_\tau.\Sigma \to Q$ is a function satisfying $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$.*

LEMMA 14. *If $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ is an experiment family, then*

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \ge \lfloor \lg_{|\Lambda|} |Q| \rfloor \,/\, (\max_{\alpha\in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))).$$

PROOF. Since $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$, the states $\mathsf{state}(\mathbf{e}_\alpha)$ are pairwise distinct and so are their encodings $\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))$. Since there are fewer than $|Q|$ strings of length strictly less than $\lfloor \lg_{|\Lambda|} |Q| \rfloor$, for

some $\alpha \in Q$ we have $\mathsf{len}(\mathsf{enc}(\mathsf{state}(\mathbf{e}_\alpha))) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor$. Then

$$\mathsf{wcmo}(\mathcal{D}_\tau, n, m) \geq \mathsf{mmo}(\mathcal{D}_\tau, \mathbb{C}_\alpha) \geq \frac{\mathsf{len}(\mathsf{state}(\mathbf{e}_\alpha))}{\mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))} \geq \frac{\lfloor \lg_{|\Lambda|} |Q| \rfloor}{\max_{\alpha' \in Q} \mathsf{len}(\mathbb{C}_{\alpha'}.\mathsf{rval}(\mathbf{e}_{\alpha'}))}. \quad \square$$

To apply this lemma to the best effect, we need to find experiment families with $|Q|$ as large as possible and $\mathsf{len}(\mathbb{C}_{\alpha'}.\mathsf{rval}(\mathbf{e}_{\alpha'}))$ as small as possible. Finding such families is challenging, as there is no systematic way to derive them. We relied on intuitions about "which situations force replicas to store a lot of information" when searching for experiment families.

Driver programs. We define experiment families using ***driver programs*** (e.g., see Fig. 11). These are written in imperative pseudocode and use traditional constructs like loops and conditionals. As they execute, they construct concrete executions of the data type library $[\tau \mapsto \mathcal{D}_\tau]$ by means of the following instructions, each of which triggers a uniquely-determined transition from Fig. 3:

$\mathsf{do}_r\ o\ ^t$	do operation o on x at replica r with timestamp t
$u \leftarrow \mathsf{do}_r\ o\ ^t$	same, but assign the return value to u
$\mathsf{send}_r(mid)$	send a message for x with identifier mid at r
$\mathsf{receive}_r(mid)$	receive the message mid at replica r

Programs explicitly supply timestamps for do and message identifiers for send and receive. We require that they do this correctly, e.g., respect uniqueness of timestamps. When a driver program terminates, it may produce a return value. For a program P, an implementation $\mathcal{D}_\tau$, and a configuration (R, T), we let $\mathsf{exec}(\mathcal{D}_\tau, (R, T), P)$ be the concrete execution of the data type library $[\tau \mapsto \mathcal{D}_\tau]$ starting in (R, T) that results from running P; we define $\mathsf{result}(\mathcal{D}_\tau, (R, T), P)$ as the return value of P in this run.

6.3 Lower Bound for State-Based Counter (ctr)

THEOREM 15. *If $\mathcal{D}_{\mathtt{ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{ctr}}$, then $\mathcal{D}_{\mathtt{ctr}}$ is $\widehat{\Omega}(n)$.*

We start by formulating a suitable experiment family.

LEMMA 16. *If $\mathcal{D}_{\mathtt{ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{ctr}}$, $n \geq 2$ and $m \geq n$ is a multiple of $(n-1)$, then tuple $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ as defined in the left column of Fig. 11 is an experiment family.*

The idea of the experiments is to force replica 1 to remember one number for each of the other replicas in the system, which then introduces an overhead proportional to n; cf. the implementation in Fig. 2(b). We show one experiment in Fig. 12. All experiments start with a common initialization phase, defined by $init$, where each of the replicas $2..n$ performs $m/(n-1)$ increments and sends a message after each increment. All messages remain undelivered until the second phase, defined by $exp(\alpha)$. There replica 1 receives exactly one message from each replica $r = 2..n$, selected using $\alpha(r)$. An experiment concludes with the read $\mathbf{e}_\alpha$ on the first replica.

The read-back works by performing separate tests for each of the replicas $r = 2..n$, defined by $test(r)$. For example, to determine which message was sent by replica 2 during the experiment in Fig. 12, the program $test(2)$: reads the counter value at replica 1, getting 12; delivers the final message by replica 2 to it; and reads the counter value at replica 1 again, getting 14. By observing the difference, the program can determine the message sent during the experiment: $\alpha(2) = 5 - (14 - 12) = 3$.

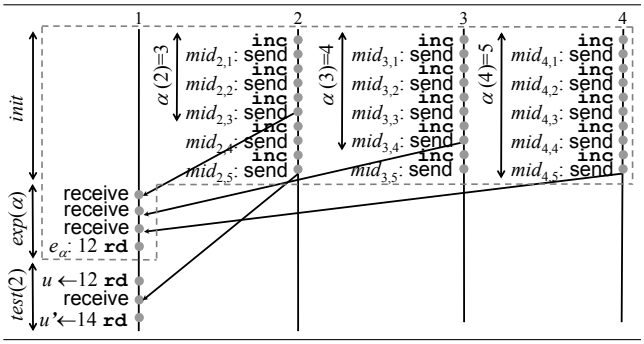
PROOF OF LEMMA 16. The only nontrivial obligation is to prove $\mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha)) = \alpha$. Let $(R_\alpha, T_\alpha) = \mathsf{final}(\mathbb{C}_\alpha)$. Then

$$\begin{aligned}
\alpha(r) &\overset{\text{(i)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_0, T_0), (init; exp(\alpha); test(r))) \\
&= \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_\alpha, T_\alpha), test(r)) \\
&\overset{\text{(ii)}}{=} \mathsf{result}(\mathcal{D}_{\mathtt{ctr}}, (R_{init}[(x,1) \mapsto R_\alpha(x,1)], T_{init}), test(r)) \\
&= \mathsf{rb}(R_\alpha(x,1))(r) \;=\; \mathsf{rb}(\mathsf{state}(\mathbf{e}_\alpha))(r),
\end{aligned}$$

Figure 11. Experiment families $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ used in the proofs of Theorem 15 (ctr) and Theorem 17 (orset)

ctr	orset
Conditions on n, m (number of replicas/updates)	
$m \geq n \geq 2$ $m \mod (n-1) = 0$	$m \geq n \geq 2$ $(m-1) \mod (n-1) = 0$
Index set Q	
$Q = ([2..n] \to [1..\frac{m}{n-1}])$	$Q = ([2..n] \to [1..\frac{m-1}{n-1}])$
Family size $\lvert Q \rvert$	
$\lvert Q \rvert = (\frac{m}{n-1})^{n-1}$	$\lvert Q \rvert = (\frac{m-1}{n-1})^{n-1}$
Driver programs	
procedure $init$ **for all** $r \in [2..n]$ **for all** $i \in [1..\frac{m}{n-1}]$ do_r inc $^{rm+i}$ $\mathsf{send}_r(mid_{r,i})$	**procedure** $init$ **for all** $r \in [2..n]$ **for all** $i \in [1..\frac{m-1}{n-1}]$ do_r add(0) $^{rm+i}$ $\mathsf{send}_r(mid_{r,i})$
procedure $exp(\alpha)$ **for all** $r \in [2..n]$ $\mathsf{receive}_1(mid_{r,\alpha(r)})$ do_1 rd $^{(n+2)m}$ // read $\mathbf{e}_\alpha$	**procedure** $exp(\alpha)$ **for all** $r \in [2..n]$ $\mathsf{receive}_1(mid_{r,\alpha(r)})$ do_1 remove(0) $^{(n+2)m}$ do_1 rd $^{(n+3)m}$ // read $\mathbf{e}_\alpha$
procedure $test(r)$ $u \leftarrow \mathsf{do}_1$ rd $^{(n+3)m}$ $\mathsf{receive}_1(mid_{r,\frac{m}{n-1}})$ $u' \leftarrow \mathsf{do}_1$ rd $^{(n+4)m}$ **return** $\frac{m}{n-1} - (u' - u)$	**procedure** $test(r)$ **for all** $i \in [1..(\frac{m-1}{n-1})]$ $\mathsf{receive}_1(mid_{r,i})$ $u \leftarrow \mathsf{do}_1$ rd $^{(n+4)m+i}$ **if** $0 \in u$ **return** $i-1$ **return** $\frac{m-1}{n-1}$
Definition of executions $\mathbb{C}_\alpha$	
$\mathbb{C}_\alpha = \mathsf{exec}(\mathcal{D}_\tau, (R_0, T_0), init; exp(\alpha))$ where $(R_0, T_0) = ([x \mapsto \mathcal{D}_\tau.\vec{\sigma}_0], \emptyset)$	
Definition of read-back function $\mathsf{rb} : \mathcal{D}_\tau.\Sigma \to Q$	
$\mathsf{rb}(\sigma) = \lambda r : [2..n].\mathsf{result}(\mathcal{D}_\tau, (R_{init}[(x,1) \mapsto \sigma], T_{init}), test(r))$ where $(R_{init}, T_{init}) = \mathsf{post}(\mathsf{exec}(\mathcal{D}_\tau, (R_0, T_0), init))$	

Figure 12. Example experiment ($n = 4$ and $m = 15$) and test for ctr. Gray dashed lines represent the configuration $(R_{init}[(x,1) \mapsto R_\alpha(x,1)], T_{init})$ where the $test$ driver program is applied.



where:

(i) This is due to $\mathcal{D}_{\mathtt{ctr}}$ $\mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]$ $\mathcal{F}_{\mathtt{ctr}}$, as we explained informally above. Let

$$C'_\alpha = \mathsf{exec}(\mathcal{D}_{\mathtt{ctr}}, (R_0, T_0), (init; exp(\alpha); test(r))).$$

Then the operation context in $\mathsf{abs}(C'_\alpha, \mathcal{V}^{\mathsf{state}})$ of the first read in $test(r)$ contains $\sum_{r=2}^{n} \alpha(r)$ increments, while that of the second read contains $(m/(n-1)) - \alpha(r)$ more increments.

(ii) We have $T_\alpha = T_{init}$ because $exp(\alpha)$ does not send any messages. Also, R_α and $R_{init}[(x,1) \mapsto R_\alpha(x,1)]$ can differ only

in the states of the replicas $2..n$. These cannot influence the run of $test(r)$, since it performs events on replica 1 only. □

PROOF OF THEOREM 15. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that m is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ given by Lemma 16. Then for any α, $\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha)$ is at most the total number of increments m in $\mathbb{C}_\alpha$. Using Lemma 14 and $m \geq n^2$, for some constants K_1, K_2, K_3, K independent from n_0, m_0 we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{ctr}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor / (\max_{\alpha \in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha))) \geq$$

$$K_1 \frac{\lg_{|\Lambda|}(\frac{m}{n-1})^{n-1}}{\mathsf{len}_{\mathbb{N}_0}(m)} \geq K_2 \frac{n \lg(m/n)}{\lg m} \geq K_3 \frac{n \lg \sqrt{m}}{\lg m} \geq Kn. \quad \square$$

6.4 Lower Bound for State-Based OR-Set (orset)

THEOREM 17. *If* $\mathcal{D}_{\mathtt{orset}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{orset}}$*, then* $\mathcal{D}_{\mathtt{orset}}$ *is* $\widehat{\Omega}(n \lg m)$.

LEMMA 18. *If* $\mathcal{D}_{\mathtt{orset}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{orset}}$*,* $n \geq 2$ *and* $m \geq n$ *is such that* $(m-1)$ *is a multiple of* $(n-1)$*, then the tuple* $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ *on the right in Fig. 11 is an experiment family.*

The proof is the same as that of Lemma 16, except for obligation (i). We therefore give only informal explanations.

The main idea of the experiments defined in the lemma is to force replica 1 to remember element instances even after they have been removed at that replica; cf. our explanation of the challenges of implementing the OR-set from §5.3. The experiments follow a similar pattern to those for `ctr`, but use different operations. In the common *init* phase, each replica $2..n$ performs $\frac{m-1}{n-1}$ operations adding a designated element 0, which are interleaved with sending messages. In the experiment phase $exp(\alpha)$, one message from each replica $r = 2..n$, selected by $\alpha(r)$, is delivered to replica 1. At the end of execution, replica 1 removes 0 from the set and performs the read $\mathbf{e}_\alpha$. The return value of this read is always the empty set.

To perform the read-back of $\alpha(r)$ for $r = 2..n$, $test(r)$ delivers all messages by replica r to replica 1 in the order they were sent and, after each such delivery, checks if replica 1 now reports the element 0 as part of the set. From $\mathcal{D}_{\mathtt{orset}}\ \mathsf{sat}[\mathcal{V}^{\mathsf{state}}, \mathsf{T\text{-}Any}]\ \mathcal{F}_{\mathtt{orset}}$ and the definition (5) of $\mathcal{F}_{\mathtt{orset}}$, we get that exactly the first $\alpha(r)$ such deliveries will have no effect on the contents of the set: the respective add operations have already been observed by the remove operation that replica 1 performed in the experiment phase. Thus, if 0 appears in the set right after delivering the i-th message of replica r, then $\alpha(r) = i - 1$, and if 0 does not appear by the time the loop is finished, then $\alpha(r) = (m-1)/(n-1)$.

PROOF OF THEOREM 17. Given n_0, m_0, we pick $n = n_0$ and some $m \geq n_0$ such that $(m-1)$ is a multiple of $(n-1)$ and $m \geq n^2$. Take the experiment family $(Q, n, m, \mathbb{C}, \mathbf{e}, \mathsf{rb})$ given by Lemma 18. For any $\alpha \in Q$, $\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha) = \emptyset$, which can be encoded with a constant length. Using Lemma 14 and $m \geq n^2$, for some constants K_1, K_2, K we get:

$$\mathsf{wcmo}(\mathcal{D}_{\mathtt{orset}}, n, m) \geq \lfloor \lg_{|\Lambda|} |Q| \rfloor \,/\, (\max_{\alpha \in Q} \mathsf{len}(\mathbb{C}_\alpha.\mathsf{rval}(\mathbf{e}_\alpha)))$$

$$\geq K_1 n \lg(m/n) \geq K_2 n \lg \sqrt{m} \geq Kn \lg m. \quad \square$$

7. Store Correctness and Consistency Axioms

Recall that we define a replicated store by a data type library $\mathbb{D}$ and a transport layer specification $\mathcal{T}$ (§4), and we specify its behavior by a function $\mathbb{F}$ from types $\tau \in \mathsf{dom}(\mathbb{D})$ to data type specifications and a set of consistency axioms (§3). The axioms are just constraints over abstract executions, such as those shown in Fig. 13; from now on we denote their sets by X. So far we have concentrated on single data type specifications $\mathbb{F}(\tau)$ and their correspondence to implementations $\mathbb{D}(\tau)$, as stated by Def. 7. In this section we consider consistency axioms and formulate the notion of correctness of the whole store $(\mathbb{D}, \mathcal{T})$ with respect to its specification $(\mathbb{F}, \mathsf{X})$.

Our first goal is to lift the statement of correctness given by Def. 7 to a store $(\mathbb{D}, \mathcal{T})$ with multiple objects of different data types. To this end, we assume a function $\mathbb{V}$ mapping each type $\tau \in \mathsf{dom}(\mathbb{D})$ to its visibility witness $\mathbb{V}$. This allows us to construct the visibility relation for a concrete execution $C \in [\![\mathbb{D}]\!] \cap \mathcal{T}$ by applying $\mathbb{V}(\tau)$ to its projection onto the events on every object of type τ:

$$\mathsf{witness}(\mathbb{V}) \;=\; \lambda C. \bigcup \{ \mathbb{V}(\mathsf{type}(x))(C|_x) \mid x \in \mathsf{Obj} \},$$

where $\cdot|_x$ projects to events over x. Then the correctness of every separate data type τ in the store with respect to $\mathbb{F}(\tau)$ according to Def. 7 automatically ensures that the behavior of the whole store is consistent with $\mathbb{F}$ in the sense of Def. 5.

PROPOSITION 19. $(\forall \tau \in \mathsf{dom}(\mathbb{D}).\ \mathbb{D}(\tau)\ \mathsf{sat}[\mathbb{V}(\tau), \mathcal{T}]\ \mathbb{F}(\tau)) \implies (\forall C \in [\![\mathbb{D}]\!] \cap \mathcal{T}.\ \mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathbb{F})$.

This motivates the following definition of store correctness. Let us write $A \models \mathsf{X}$ when the abstract execution A satisfies the axioms X.

DEFINITION 20. *A store* $(\mathbb{D}, \mathcal{T})$ *is* ***correct*** *with respect to a specification* $(\mathbb{F}, \mathsf{X})$*, if for some* $\mathbb{V}$*:*

(i) $\forall \tau \in \mathsf{dom}(\mathbb{D}).\ (\mathbb{D}(\tau)\ \mathsf{sat}[\mathbb{V}(\tau), \mathcal{T}]\ \mathbb{F}(\tau))$*; and*
(ii) $\forall C \in [\![\mathbb{D}]\!] \cap \mathcal{T}.\ (\mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathsf{X})$.

We showed how to discharge (i) in §5. The validity of axioms X required by (ii) most often depends on the transport layer specification $\mathcal{T}$: e.g., to disallow the anomaly (2) from §1, $\mathcal{T}$ needs to provide guarantees on how messages pertaining to different objects are delivered. However, data type implementations can also enforce axioms by putting enough information into messages: e.g., implementations correct with respect to $\mathcal{V}^{\mathsf{state}}$ from §4 ensure that vis is transitive regardless of the behavior of the transport layer. Fortunately, to establish (ii) in practice, we do not need to consider the internals of data type implementations in $\mathbb{D}$—just knowing the visibility witnesses used in the statements of their correctness is enough, as formulated in the following definition.

DEFINITION 21. *A set* W *of visibility witnesses and a transport layer specification* $\mathcal{T}$ ***validate*** *axioms* X*, if*

$$\forall C, \mathbb{V}.\ (C \in \mathcal{T}) \wedge (\{\mathbb{V}(\tau) \mid \tau \in \mathsf{dom}(\mathbb{V})\} \subseteq W) \implies (\mathsf{abs}(C, \mathsf{witness}(\mathbb{V})) \models \mathsf{X}).$$

Since visibility witnesses are common to wide classes of data types (e.g., state- or op-based), our proofs of the validity of axioms will not have to be redone if we add new data type implementations to the store from a class already considered.

We next present axioms formalizing several variants of eventual consistency used in replicated stores (Fig. 13 and 14) and W and $\mathcal{T}$ that validate them. We then use this as a basis for discussing connections with weak shared-memory models. Due to space constraints, we defer technical details and proofs to [12, §D].

Basic eventual consistency. EVENTUAL and THINAIR define a weak form of eventual consistency. EVENTUAL ensures that an event cannot be invisible to infinitely many other events on the same object and thus implies (1) from §1: informally, if updates stop, then reads at all replicas will eventually see all updates and will return the same values (§3.2). However, EVENTUAL is stronger than quiescent consistency: the latter does not provide any guarantees at all for executions with infinitely many updates to the store, whereas our specification implies that the return values are computed according to $\mathbb{F}(\tau)$ using increasingly up-to-date view of the store state. We formalize these relationships in [12, §D].

THINAIR prohibits values from appearing "out-of-thin-air" [28], like 42 in Fig. 14(a) (recall that registers are initialized to 0). Cycles in $\mathsf{ro} \cup \mathsf{vis}$ that lead to out-of-thin-airs usually arise

Figure 13. A selection of consistency axioms over an execution $(E, \mathsf{repl}, \mathsf{obj}, \mathsf{oper}, \mathsf{rval}, \mathsf{ro}, \mathsf{vis}, \mathsf{ar})$

Auxiliary relations

$\mathsf{sameobj}(e, f) \iff \mathsf{obj}(e) = \mathsf{obj}(f)$

Per-object causality (aka happens-before) order:

$\mathsf{hbo} = ((\mathsf{ro} \cap \mathsf{sameobj}) \cup \mathsf{vis})^+$

Causality (aka happens-before) order: $\mathsf{hb} = (\mathsf{ro} \cup \mathsf{vis})^+$

Axioms

EVENTUAL:

$\forall e \in E. \neg(\exists \text{ infinitely many } f \in E. \mathsf{sameobj}(e, f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f))$

THINAIR: $\mathsf{ro} \cup \mathsf{vis}$ is acyclic

POCV (Per-Object Causal Visibility): $\mathsf{hbo} \subseteq \mathsf{vis}$

POCA (Per-Object Causal Arbitration): $\mathsf{hbo} \subseteq \mathsf{ar}$

COCV (Cross-Object Causal Visibility): $(\mathsf{hb} \cap \mathsf{sameobj}) \subseteq \mathsf{vis}$

COCA (Cross-Object Causal Arbitration): $\mathsf{hb} \cup \mathsf{ar}$ is acyclic

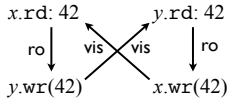
Figure 14. Anomalies allowed or disallowed by different axioms

(a) Disallowed by THINAIR:

$x, y : \texttt{intreg}$

$i = x.\texttt{rd} \;\|\; j = y.\texttt{rd}$

$y.\texttt{wr}(i) \;\|\; x.\texttt{wr}(j)$

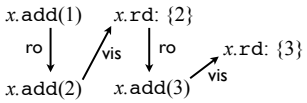


(b) Disallowed by POCV:

$x : \texttt{orset}$

$x.\texttt{add}(1) \;\|\; i = x.\texttt{rd} \;\|\; j = x.\texttt{rd}$

$x.\texttt{add}(2) \;\|\; x.\texttt{add}(3)$

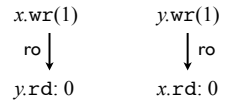


(c) Allowed by COCV and COCA:

$x, y : \texttt{intreg}$

$x.\texttt{wr}(1) \;\|\; y.\texttt{wr}(1)$

$i = y.\texttt{rd} \;\|\; j = x.\texttt{rd}$



from effects of speculative computations, which are done by some older replicated stores [36].

THINAIR is validated by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and T-Any, and EVENTUAL by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and the following condition on C ensuring that every message is eventually delivered to all other replicas and every operation is followed by a message generation:

$$(\forall e \in C.E.\, \forall r, r'.\, C.\mathsf{act}(e) = \mathsf{send} \wedge C.\mathsf{repl}(e) = r \wedge r \neq r' \implies \exists f.\, C.\mathsf{repl}(f) = r' \wedge e \xrightarrow{\mathsf{del}(C)} f) \wedge$$
$$(\forall e \in C.E.\, C.\mathsf{act}(e) = \mathsf{do} \implies \exists f.\, \mathsf{act}(f) = \mathsf{send} \wedge e \xrightarrow{\mathsf{roo}(C)} f),$$

where $\mathsf{roo}(C)$ is $\mathsf{ro}(C)$ projected to events on the same object.

Causality guarantees. Many replicated stores achieve availability and partition tolerance while providing stronger guarantees, which we formalize by the other axioms in Fig. 13. We call an execution ***per-object***, respectively, ***cross-object causally consistent***, if it is eventually consistent (as per above) and satisfies the axioms POCV and POCA, respectively, COCV and COCA. POCV guarantees that an operation sees all operations connected to it by a causal chain of events on the same object; COCV also considers causal chains via different objects. Thus, POCV disallows the execution in Fig. 14(b), and COCV the one in §3.1, corresponding to (2) from §1. POCA and COCA similarly require arbitration to be consistent with causality. The axioms highlight the principle of formalizing stronger consistency models: including more edges into vis and ar, so that clients have more up-to-date information.

Cross-object causal consistency is implemented by, e.g., COPS [27] and Gemini [23]. It is weaker than strong consistency, as it allows reading stale data. For example, it allows the execution in Fig. 14(c), where both reads fetch the initial value of the register, despite writes to it by the other replica. It is easy to check that this outcome cannot be produced by any interleaving of the events at the two replicas, and is thus not strongly consistent.

An interesting feature of per-object causal consistency is that state-based data types ensure most of it just by the definition of $\mathcal{V}^{\mathsf{state}}$: POCV is validated by $\{\mathcal{V}^{\mathsf{state}}\}$ and T-Any. If the witness set is $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$, then we need $\mathcal{T}$ to guarantee the following: informally, if a send event e and another event f are connected by a causal chain of events on the same object, then the message created by e is delivered to $C.\mathsf{repl}(f)$ by the time f is done. POCA is validated by $\{\mathcal{V}^{\mathsf{state}}, \mathcal{V}^{\mathsf{op}}\}$ and the transport layer specification $(\mathsf{roo}(C) \cup \mathsf{del}(C))^+|_{\mathsf{do}} \subseteq \mathsf{ar}(C)$. This states that timestamps of events on every object behave like a Lamport clock [22]. Conditions for COCV and COCA are similar.

There also exist consistency levels in between basic eventual consistency and per-object causal consistency, defined using so-called ***session guarantees*** [35]. We cover them in [12, §D].

Comparison with shared-memory consistency models. Interestingly, the specializations of the consistency levels defined by the axioms in Fig. 13 to the type `intreg` of LWW-registers are very close to those adopted by the memory model in the 2011 C and C++ standards [5]. Thus, POCA and POCV define the semantics of the fragment of C/C++ restricted to so-called *relaxed* operations; there this semantics is defined using *coherence* axioms, which are analogous to session guarantees [35]. COCV and COCA are close to the semantics of C/C++ restricted to *release-acquire* operations. However, C/C++ does not have an analog of EVENTUAL and does not validate THINAIR, since it makes the effects of speculations visible to the programmer [4]. We formalize the correspondence to C/C++ in [12, §D]. In the future, this correspondence may open the door to applying technology developed for shared-memory models to eventually consistent systems; promising directions include model checking [3, 9], automatic inference of required consistency levels [26] and compositional reasoning [4].

8. Related Work

For a comprehensive overview of replicated data type research we refer the reader to Shapiro et al. [32]. Most papers proposing new data type implementations [6, 31–33] do not provide their formal declarative specifications, save for the expected property (1) of quiescent consistency or first specification attempts for sets [6, 7]. Formalizations of eventual consistency have either expressed quiescent consistency [8] or gave low-level operational specifications [17].

An exception is the work of Burckhardt et al. [10, 13], who proposed an axiomatic model of causal eventual consistency based on visibility and arbitration relations and an operational model based on revision diagrams. Their store specification does not provide customizable consistency guarantees, and their data type specifications are limited to the sequential $\mathcal{S}$ construction from §3.2, which cannot express advanced conflict resolution used by the multi-value register, the OR-set and many other data types [32]. More significantly, their operational model does not support general op- or state-based implementations, and is thus not suited for studying the correctness or optimality of these commonly used patterns.

Simulation relations have been applied to verify the correctness of sequential [25] and shared-memory concurrent data type implementations [24]. We take this approach to the more complex setting of a replicated store, where the simulation needs to take into account multiple object copies and messages and associate them with structures on events, rather than single abstract states. This poses technical challenges not considered by prior work, which we address by our novel notion of replication-aware simulations.

The distributed computing community has established a number of asymptotic lower bounds on the complexity of implementing certain distributed or concurrent abstractions, including one-

shot timestamp objects [20] and counting protocols [15, 30]. These works have considered either programming models or metrics significantly different from ours. An exception is the work of Charron-Bost [14], who proved that the size of vector clocks [29] is optimal to represent the happens-before relation of a computation (similar to the visibility relation in our model). Specifications of `mvr` and `orset` rely on visibility; however, Charron-Bost's result does not directly translate into a lower bound on their implementation complexity, since a specification may not require complete knowledge about the relation and an implementation may represent it in an arbitrary manner, not necessarily using a vector.

9. Conclusion and Future Work

We have presented a comprehensive theoretical toolkit to advance the study of replicated eventually consistent stores, by proposing methods for (1) specifying the semantics of replicated data types and stores abstractly, (2) verifying implementations of replicated data types, and (3) proving that such implementations have optimal metadata overhead. By proving both correctness and optimality of four nontrivial data type implementations, we have demonstrated that our methods can indeed be productively applied to the kinds of patterns used by practitioners and researchers in this area.

Although our work marks a big step forward, it is only a beginning, and creates plenty of opportunities for future research. We have already made the first steps in extending our specification framework with more features, such as mixtures of consistency levels [23] and transactions [34, 37]; see [11]. In the future we would also like to study more data types, such as lists used for collaborative editing [32], and to investigate metadata bounds for data type implementations other than state-based ones, including more detailed overhead metrics capturing optimizations invisible to the worst-case overhead analysis. Even though our execution model for replicated stores follows the one used by replicated data type designers [33], there are opportunities for bringing it closer to actual implementations. Thus, we would like to verify the algorithms used by store implementations [27, 34, 37] that our semantics abstracts from. This includes fail-over and session migration protocols, which permit clients to interact with multiple physical replicas, while being provided the illusion of a single virtual replica.

Finally, by bringing together prior work on shared-memory models and data replication, we wish to promote an exchange of ideas and results between the research communities of programming languages and verification on one side and distributed systems on the other.

Acknowledgements. We thank Hagit Attiya, Anindya Banerjee, Carlos Baquero, Lindsey Kuper and Marc Shapiro for comments that helped improve the paper. Gotsman was supported by the EU FET project ADVENT, and Yang by EPSRC.

References

[1] Riak key-value store. http://basho.com/products/riak-overview/.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12), 1996.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, 2013.

[4] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[5] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[6] A. Bieniusa, M. Zawirski, N. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. Technical Report 8083, INRIA, 2012.

[7] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *DISC*, 2012.

[8] A.-M. Bosneag and M. Brockmeyer. A formal model for eventual consistency semantics. In *IASTED PDCS*, 2002.

[9] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[10] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *ECOOP*, 2012.

[11] S. Burckhardt, A. Gotsman, and H. Yang. Understanding eventual consistency. Technical Report MSR-TR-2013-39, Microsoft Research, 2013.

[12] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality (extended version), 2013. http://research.microsoft.com/apps/pubs/?id=201602.

[13] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[14] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1), 1991.

[15] J.-Y. Chen and G. Pandurangan. Optimal gossip-based aggregate computation. In *SPAA*, 2010.

[16] N. Conway, R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *SOCC*, 2012.

[17] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. In *PODC*, 1996.

[18] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

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

[20] M. Helmi, L. Higham, E. Pacheco, and P. Woelfel. The space complexity of long-lived and one-shot timestamp implementations. In *PODC*, 2011.

[21] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1, 1972.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

[23] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.

[24] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

[25] B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Symposium on Very High Level Languages*, 1974.

[26] F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, 2012.

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

[28] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[29] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1989.

[30] S. Moran, G. Taubenfeld, and I. Yadin. Concurrent counting. In *PODC*, 1992.

[31] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[32] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

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

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

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

[36] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[37] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. Preguiça. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. Technical Report 8347, INRIA, 2013.