HAMBAND: RDMA Replicated Data Types*

Farzin Houshmand†
University of California, Riverside
Riverside, California, USA
fhous001@ucr.edu

Javad Saberlatibari†
University of California, Riverside
Riverside, California, USA
jsabe004@ucr.edu

Mohsen Lesani
University of California, Riverside
Riverside, California, USA
lesani@cs.ucr.edu

Abstract

Data centers are increasingly equipped with RDMAs. These network interfaces mark the advent of a new distributed system model where a node can directly access the remote memory of another. They have enabled microsecond-scale replicated services. The underlying replication protocols of these systems execute all operations under strong consistency. However, strong consistency can hinder response time and availability, and recent replication models have turned to a hybrid of strong and relaxed consistency. This paper presents RDMA well-coordinated replicated data types, the first hybrid replicated data types for the RDMA network model. It presents a novel operational semantics for these data types that considers three distinct categories of methods and captures their required coordination, and formally proves that they preserve convergence and integrity. It implements these semantics in a system called HAMBAND that leverages direct remote accesses to efficiently implement the required coordination protocols. The empirical evaluation shows that HAMBAND outperforms the throughput of existing message-based and strongly consistent implementations by more than 17x and 2.7x respectively.

CCS Concepts: • **Software and its engineering → Formal language definitions**; **Correctness**; **Semantics**; **Consistency**; • **Computer systems organization → Reliability**; **Availability**.

Keywords: RDMA, Operational Semantics, WRDT

ACM Reference Format:
Farzin Houshmand, Javad Saberlatibari, and Mohsen Lesani. 2022. HAMBAND: RDMA Replicated Data Types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523426

*Supported by the National Science Foundation under grant 1942711.
†The two authors contributed equally to this research.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523426

1 Introduction

Data centers equipped with RDMA network interfaces are pervasive. These network interfaces support Remote Direct Memory Access (RDMA) [2, 46] from one node to another without going through the network and operating system stack or requiring CPU cycles from the other node. RDMAs mark the advent of a new model for distributed computing that combines the traditionally separate models of shared memory and message-passing, and have motivated new protocol designs [5, 6, 78]. This technology has been used to enable microsecond-scale [14] replicated services whose availability and low-latency are critical in applications such as finance and control.

RDMAs have been utilized to accelerate key-value stores [32, 45] and transactions [47, 87, 88]. In particular, they have been used to implement State Machine Replication (SMR) [79]. At its core, an SMR is a consensus or atomic broadcast protocol that executes requests in the same total order across replicas, and provides strong consistency. From the long-lasting class of SMR protocols and systems, RDMA-accelerated SMRs have gained recent attention in projects such as DARE [74], APUS [86], Derecho [41], HovercRaft [49], Mu [7], Hermes [48] and Kite [35]. In contrast to traditional message-passing SMRs whose latencies are hundreds of microseconds, RDMA SMRs exhibit latencies that are less than a dozen microseconds. To maintain the low latency, it is crucial to avoid overloading the system. Therefore, the throughput of RDMA replicated systems is an important factor for their responsiveness as well [34].

In the message-passing model, SMR protocols such as Viewstamp [69], Paxos [52], Raft [70] and Spanner [29] provide strong consistency and have been the de facto standard for replication. However, practitioners [3, 28, 51, 72, 84] soon realized that SMR does not provide enough throughput, responsiveness and availability [4, 19, 20] for industrial applications, and opted for relaxed notations of consistency. In fact, deployments of SMR are often limited to small cluster sizes [25, 29, 40]. The large class of relaxed consistency notions [80] can be more efficiently provided [53, 73]. However, these notions forgo the total order of operations across replicas. Therefore, an immediate question is the safety of these notions for replication. Convergent and Commutative Replicated Data Types (CRDTs) [81, 82] and similar notions [8, 77] formally define replicated data types that converge under relaxed consistency. In addition to convergence, RA-linearizability [85] and ACC [59] define

specifications for the functional correctness of these types. The definition of these types and their specifications led to projects on their composition [27, 61, 89, 89], and verification [18, 24, 33, 36, 60, 65, 91]. They were later followed by more expressive convergent types such as cloud, mergeable and reactive types [22, 23, 44, 64]. Convergence might be enough for simple objects such as counters. However, relaxed consistency can further violate the integrity [10] of objects (such as maintaining a non-negative balance for a bank account). Thus, replicated data types that preserve integrity under relaxed consistency [9, 67, 90] followed.

However, not all operations can preserve convergence and integrity under relaxed consistency. Some operations do need strong consistency. Therefore, several projects considered hybrid models where each operation is executed under either relaxed or strong consistency based on its semantics. These projects include IPA [11], Sieve [55–57], Indigo [12, 13], CISE [37], Quelea [83], Carol [54] Hamsaz [39] and ECRO [30]. Hamsaz analyzes the given object to find the conflicting and dependent pairs of methods. It then synthesizes well-coordinated replicated objects that synchronize for conflicting, and preserve dependencies for dependent method calls. Well-coordinated replicated data types (WRDTs) guarantee convergence and integrity. Hampa [58] later added recency guarantees. Other projects tested and verified [15, 17, 21, 43, 66, 75], and repaired [76] replicated objects in hybrid models. Yet, others [38, 50, 62] considered the flow between relaxed and strong consistency.

However, the distributed system model that was considered for CRDTs and WRDTs was always the traditional message-passing network model. What is the semantics of CRDTs and WRDTs in the RDMA network model? What are the efficient coordination protocols that can leverage RDMAs to implement CRDTs and WRDTs?

RDMA offers two classes of communication primitives: two-sided and one-sided. Two-sided communication has similar semantics to the traditional message-passing model. A node can execute a send operation to communicate a message to another node. The other node should execute an explicit receive operation to deliver and process the message. On the other hand, one-sided communication has similar semantics to the traditional shared memory model. A node directly performs a write or read operation on the memory of another node. The access is performed without involving the CPU of the other node. The new class, one-sided communication, tends to deliver lower response time since it bypasses the network and operating system stack and does not interrupt the CPU of the other node. How can well-coordination be efficiently implemented by one-sided communication?

This paper presents a novel operational semantics for RDMA WRDTs. The semantics divides methods of a given object into three categories, reducible, irreducible conflict-free, and conflicting, and declares distinct coordination requirements for each. The semantics does not perform any message-passing. In particular, reducible method calls can be performed with a single one-sided write operation that can be executed in parallel on the replicas. Similarly, the coordination for the other two categories is a sequence of local operations followed by one-sided remote operations. Further, we define an abstract operational semantics for WRDTs that captures well-coordination conditions. We prove that the concrete semantics of RDMA WRDTs refines the abstract semantics of WRDTs. Therefore, any execution of an RDMA WRDT is well-coordinated. Since (op-based) CRDTs are a special instance of WRDTs, each of the above two WRDT semantics subsume the semantics of CRDTs.

The operational semantics of RDMA WRDTs serves as a specification for their implementation and runtime system. We implement RDMA WRDT on top of consensus and reliable broadcast abstractions for the RDMA network model. We adopted and implemented several CRDTs, and WRDTs. We evaluated our implementations by comparing them to both message-based and SMR-based implementations. The results show that on average, WRDTs exhibit more than 17x and 2.7x higher throughput respectively with almost the same response time.

In summary, this paper makes the following contributions:

- It introduces RDMA WRDTs, the first hybrid replicated data types for the RDMA network model.
- It presents a novel operational semantics for RDMA WRDTs that is based solely on one-sided communication. It divides methods to three categories and defines the required coordination for each.
- It captures the notion of well-coordination as the abstract WRDT operational semantics, and formally proves that the RDMA WRDT semantics refine the abstract WRDT semantics, and preserve integrity and convergence.
- It efficiently implements the coordination protocols for RDMA WRDTs using only one-sided communication.
- It empirically shows that the RDMA WRDTs outperform the throughput of the existing message-based and SMR-based implementations.

2 Overview

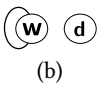
We now illustrate RDMA replication with the familiar bank account example.

Example. As Figure 1.(a) shows, an object of the *Account* class stores the balance state b, with the integrity property (or invariant) $\mathcal{I}$ that requires the balance to stay non-negative. The class exposes the two update methods deposit and withdraw, which return the updated balance state, and the query method balance that returns the current balance.

The goal is to replicate an object on the given set of host processes such that the processes can issue requests to call update and query methods on the object. The processes should coordinate the calls so that the *integrity* property

```
class Account
    b : Int
    I := λb. b ≥ 0
    update deposit (a)
        b + a
    update withdraw (a)
        b − a
    query balance ()
        b
```

(a)



(b)

(c)

Figure 1. Bank Account. (a) The user specification, (b) The conflict graph, and (c) The dependency graph

is always preserved, and the states of the processes eventually *converge*.

Well-coordination. A withdraw call issued at a process should be *locally permissible*: it should not overdraft the account, *i.e.*, not violate the invariant. Further, preserving integrity and convergence requires enforcing certain orders between calls across processes. For example, consider two withdraw calls, that each zeros the balance, are concurrently executed at two processes, and are propagated and applied to the other process in the opposite order. The second withdraw call in each process overdrafts the account and violates the integrity property. Although it was locally permissible in its issuing process, it becomes impermissible in the other process. We say that two withdraw calls *permissible-conflict*. Further, consider a withdraw call that zeros the balance is executed right after a deposit call in a process. If the withdraw call is propagated and applied to other processes before the deposit call, then the withdraw call overdraft the account in the other processes. We say that the withdraw call is *dependent* on the deposit call. Similarly, for a set object, if an *add* and a *remove* call on the same element concurrently execute on two processes, and are applied to the other process in the opposite order, the state of the set object diverges. We say that the two calls *add* and *remove state-conflict*.

A replicated execution is well-coordinated if it is (1) locally permissible: every call should be permissible in the issuing process, (2) conflict-synchronizing: any pair of conflicting calls should have the same order across processes, and (3) dependency-preserving: a received call should be applied locally only if all the calls that it succeeded in the issuing process and is dependent on are already applied.

RDMA Coordination. RDMA allows a process to directly access the memory of other processes. Direct reads and writes are considerably faster than reading and writing by message-passing through the network stack. How can RDMA-enabled processes provide well-coordinated replicated objects? How can direct memory accesses accelerate the required coordination? Coordination mechanisms that can be captured as *local accesses and then a sequence of independent remote accesses* can execute efficiently. In these mechanisms, an access does not need to wait for a round-trip to receive the result of the previous access. In Figure 2, we showcase the coordination of RDMA replicated objects for our account example with three processes. Each process stores the state σ of the object: in our example, the balance for the account. It further stores other pieces of state that we visit in turn.

Conflicting methods. The conflict relation between the methods of an object induces the conflict graph. As Figure 1.(b) shows, in our account example, the conflict graph has a loop on the withdraw method, and the deposit method is conflict-free. Every pair of calls on adjacent methods of the conflict graph need to be synchronized to have the same order in all the processes. We call a connected component of the conflict graph a *synchronization group*. Each synchronization group will have a leader process. Every other process in the group is a follower. Each follower process stores a *buffer* L for each synchronization group that stores pending calls on the methods of that group. In the account example, there is a group for withdraw method calls. In Figure 2, the leader for this group is p_1 and each follower process keeps a buffer L for the withdraw calls. The leader checks local permissibility, orders and locally applies calls on the methods of the group, and remotely appends the ordered calls to the buffers of the followers. A follower process periodically traverses its buffer and applies the pending calls to the state σ.

Follower processes receive updates without actively listening for and receiving messages through the network stack. The buffer for a group at each follower process is written by only the leader of the group and is read by only that local process. Therefore, the leader itself maintains the tail pointer of the buffer, and the coordination operation of the leader is locally reading and updating the tail pointer, and then remotely writing the call. We will see more details about how the buffers are managed in section 4.

Dependencies. As we saw above, the dependencies of calls should be respected. Therefore, each process keeps a mapping A from each process p and method u to the number of calls on u from p that are locally applied. When a call is shipped to be appended to a remote buffer, it is shipped with an account of its dependencies. The *dependency map* D of a call on a method u is the projection of the *applied map* A of the issuing process over the methods that u is dependent on. To respect the dependencies, when a process traverses a buffer, it applies a call only if the local applied map A is *pointwise greater than* the dependency map D that accompanies the call. When a call is applied, the local applied map A is advanced for the issuing process. As Figure 1.(c) shows, in our account example, the dependencies of the withdraw method is the singleton set containing the deposit method. On the other hand, the deposit method is dependence-free. Thus, the applied and dependency maps are reduced to arrays indexed by process identifiers that store the number of deposit calls issued by each process.

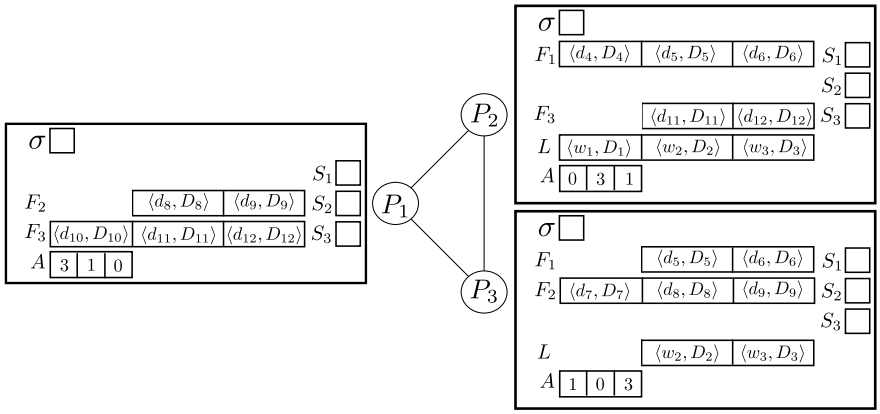


Figure 2. RDMA Replicated Bank Account

Conflict-free methods. Calls to conflict-free methods such as the deposit method can avoid synchronization. Each process can autonomously propagate its conflict-free methods to other processes. Each process stores a *buffer* F for conflict-free calls from each other process. A process p that issues a conflict-free call checks that it is locally permissible, applies it locally, and remotely appends it to the buffers that other processes store for p. For example, in Figure 2, the buffer F_2 in process p_1 stores the deposit calls issued by the process p_2. Similarly, the process p_3 stores a buffer F_2 for deposit calls issued by p_2. Similar to a conflicting call, a conflict-free call is accompanied by its dependency map D and is applied only if the local applied map A is ahead of its dependency map.

A process stores a buffer of conflict-free calls for each other process. The other process is the only writer of the buffer and the local process is the only reader. Therefore, the other process can perform the update by locally reading and writing the buffer tail, and then remotely writing the call (and its dependencies). Sharing buffers would require synchronization across processes. RDMA does provide compare-and-swap operations; however, they are more expensive than reads and writes and we avoid them with a *single-writer* design.

Reducible methods. We saw that conflict-free calls can avoid synchronization. The issuing process can simply propagate them to other processes by remote writes. An important observation is that for some conflict-free calls, even processing the calls can be done at the issuing process, and the other processes can receive the updates with no effort. In our account example, two deposit calls can be *summarized* to a single deposit with an amount equal to the sum of the two amounts. Therefore, the issuing process can summarize its deposit calls into a single deposit call, and remotely write only that call. As Figure 2 shows, each buffer of conflict-free calls F can be replaced with a single summary call S. Each process stores a summary call for itself and each other process. When a process issues a new deposit call, it first calculates the summary of its current summary and the new call. It then overwrites the summary locally for itself and remotely for each other process. It further advances the applied map for the current process both locally and remotely. In contrast to buffers, each process keeps its own summary since the process needs its own summary to recalculate it.

Up to this point, the query method balance would simply return the stored state σ. However, in the presence of summarized calls, it should apply all the locally stored summary calls to σ to calculate the current state. In our example, it should apply the calls S_1, S_2 and S_3 to σ. Since the summary calls are conflict-free, they can be applied in any order. In a more elaborate object, it might be possible to summarize only separate subsets of methods which we call *summarization groups*. Each process stores a summary call from each process per summarization group.

Method categories. Summarization can accelerate the propagation and processing of calls. The summary is locally recalculated and is propagated by a *single remote write*. The caveat is that not all summarizable methods can be propagated as above. The method needs to be not only conflict-free but also dependence-free. If a method call has dependencies and it is summarized and remotely written for another process, the other process might not have applied the dependencies yet. Therefore, we consider three categories of methods. We say that a method is ***reducible*** as described above only if it is conflict-free, dependence-free and summarizable. We call a method ***irreducible conflict-free*** if it is conflict-free but either not summarizable or not dependence-free. For example, in a grow-only set that has a contains and an add method (to add an element but not a set), the method add is conflict-free but is not summarizable. On the other hand, if the set object has an add method to add a set, then the add method is summarizable. As another example, consider a bank that is represented as a map that associates accounts to their balances, and in addition to deposit and withdraw,

σ	:	Σ	State
$\mathcal{I}$			Invariant (Integrity)
v	:	V	Value
u	:	U	Update Method
q	:	Q	Query Method
d	:	$\lambda x, \sigma.\ e$	Definition
e			Expression
o	$:=$	$\langle \Sigma, \mathcal{I}, \overline{u := d}, \overline{q := d} \rangle$	Object
p	:	P	Process
r	:	R	Request Identifier
$c : C$	$:=$	$u(v)_{p,r}$	Update Method Call
a	$:=$	$q(v)$	Query Method Call
ℓ	$:=$	$(p, u(v)_r) \mid (p, q(v))$	Label
τ	$:=$	ℓ^*	Trace

Figure 3. Basic Syntax

exposes the open method to open accounts. The deposit method is conflict-free but is dependent on the open method. Irreducible conflict-free methods use the conflict-free buffers that we saw above. Finally, ***conflicting*** calls use the conflicting buffers. We will consider these categories in more detail and the semantics of RDMA replicated objects in the next section.

3 Replicated Data Types

In this section, we first present how the high-level specifications of object data types can be captured. We then present a core operational semantics for well-coordinated replicated data types (WRDTs), and prove that it guarantees integrity and convergence. This abstract semantics will serve as the specification for replicated data types. We then present the semantics of RDMA replicated data types. It divides the methods of an object into three categories, and declares separate coordination requirements for each. We prove that this concrete semantics refines the earlier abstract semantics of WRDTs, and therefore, guarantees integrity and convergence.

3.1 Object Data Types

As Figure 3 shows, a class of objects is a tuple $\langle \Sigma, \mathcal{I}, \overline{u := d}, \overline{q := d} \rangle$ that defines the state type Σ, the invariant (or integrity property) $\mathcal{I}$ on the state, and the definitions of update methods u and query methods q. The invariant (or integrity) $\mathcal{I}$ is a predicate on the state. The definition of an update method is a function from the parameter and the pre-state σ to the post-state. Similarly, the definition of a query method is a function from the parameter and the pre-state σ to the return value. The object is replicated on the set of processes P. Any process p can accept and issue update calls $u(v)$ or query calls $q(v)$. The calls have unique request identifiers r. An update call is decorated with the issuing process p and the request identifier r. We elide these decorations when they are not needed or are evident from the context.

ss	:	$P \mapsto \Sigma$	Replicated State
xs	:	$P \mapsto \mathsf{List}(C)$	Replicated Execution
W	$:=$	$\langle ss, xs \rangle$	World
W_0	$:=$	$\langle \overline{[p \mapsto \sigma_0]}_{p \in P}, \overline{[p \mapsto \emptyset]}_{p \in P} \rangle$	Initial World

Figure 4. WRDT State

Clients can request method calls at every process, and the processes coordinate these calls. A label for a call request is the pair of the issuing process and a call, and a trace is a sequence of labels.

3.2 Semantics of Well-Coordinated Replicated Data Types

We now present the operational semantics for well-coordinated replicated data types (WRDTs). We first see the replicated state and the coordination conditions that the transition rules use.

Replicated State. The state of the given object is replicated across processes. The replicated state ss is a mapping from each process p to its states σ. The execution history x of a process is modeled as a permutation of a set of calls. Since query calls do not mutate the state, an execution history only keeps update calls. We write $c \in x$ to denote that the call c is in the history x. An execution history x defines a total order on its calls: we write $c \prec_x c'$ iff the call c precedes the call c' in the execution history x. A replicated execution xs is a function from each process to its execution history. The state W of our operational semantics is the pair of the replicated state ss and the replicated execution xs. In the initial state W_0, the state of each process is the same state σ_0 that satisfies the invariant $\mathcal{I}$, and the history of each process is an empty list.

Coordination Conditions. We now define the coordination conditions in steps. For the sake of brevity, we elide separate definition environments.

State-conflict. A replicated execution is convergent if the state of the processes is the same after all the calls are propagated to all processes. Out of order delivery of calls at different processes can lead to divergence of their states. For example, for the set data type, according to the execution order of the two calls add and remove of the same element, there are two possible resulting states. Therefore, they should synchronize. Two method calls c_1 and c_2 $\mathcal{S}$-commute, written as $c_1 \blacklozenge_{\mathcal{S}} c_2$, iff $c \circ c' = c' \circ c$ (where $\circ$ is function composition). Otherwise, they $\mathcal{S}$-conflict, written as $c_1 \bowtie_{\mathcal{S}} c_2$.

Integrity and Permissibility. In the execution history of a process, the post-state of a call is the pre-state of the next call. The body of each method can assume and rely on the invariant in the pre-state; it should then preserve it in the

post-state for the next call. The notion of permissibility requires the invariant to hold in the post-state. A method call c is permissible in a state σ, written as $\mathcal{P}(\sigma, c)$, iff $\mathcal{I}(c(\sigma))$. The initial state is assumed to satisfy the invariant. Therefore, if it is ensured that every call is permissible in its pre-state, then by induction, every call enjoys integrity in its pre-state. Permissibility leads to integrity.

Invariant-sufficiency. There are calls (*e.g.*, deposit on a bank account) that are always permissible (*i.e.*, a deposit never overdrafts) as far as they are applied to a state that has integrity. Thus, when they are broadcast and executed on another process, in order to be permissible, they only need the pre-state to have integrity. A call c is invariant-sufficient iff for every state σ, if $\mathcal{I}(\sigma)$ then $\mathcal{P}(\sigma, c)$.

Permissible-Right-Commutativity. However, not all calls are invariant-sufficient. For example, a withdraw call may be permissible in a process but may become impermissible in another where it is executed after a racing withdraw call. A call c_1 $\mathcal{P}$-R-commutes with another c_2, written as $c_1 \triangleright_{\mathcal{P}} c_2$, iff for every state σ, if $\mathcal{P}(\sigma, c_1)$ then $\mathcal{P}(c_2(\sigma), c_1)$. *i.e.*, permissibility holds even after c_1 is pushed right after c_2.

Permissible-conflict. We say that c_1 $\mathcal{P}$-concurs with a call c_2, if c_1 is invariant-sufficient or $c_1 \triangleright_{\mathcal{P}} c_2$. Otherwise, c_1 $\mathcal{P}$-conflicts with c_2 and needs to synchronize with it.

Conflict. We say that two calls c_1 and c_2 *concur* iff they both $\mathcal{S}$-commute and $\mathcal{P}$-concur with each other. Otherwise, we say they *conflict* written as $c_1 \bowtie c_2$. Conflicting calls need synchronization. A call is conflict-free if it does not conflict with any other call.

Permissible-Left-Commutativity. We saw above that invariant-sufficient calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that precede them. For example, a withdraw call may be dependent on the money deposited by a preceding deposit call in the issuing process; after propagation to another process, if the withdraw moves to the left of the deposit, the withdraw call can overdraft. A call c_2 $\mathcal{P}$-L-commutes a call c_1, written as $c_2 \triangleleft_{\mathcal{P}} c_1$ iff for every state σ, if c_2 is permissible in the post-state of the call c_1 on σ, *i.e.*, $\mathcal{P}(c_1(\sigma), c_2)$, then c_2 is permissible in σ, *i.e.*, $\mathcal{P}(\sigma, c_2)$, as well.

Dependency. A call c_2 is independent of c_1, written as $c_2 \perp\!\!\!\perp c_1$, if c_2 is invariant-sufficient or $c_2 \triangleleft_{\mathcal{P}} c_1$. Otherwise, c_2 is dependent on c_1, written as $c_2 \not\perp\!\!\!\perp c_1$. If c_1 is executed before c_2 in the issuing process of c_2, and $c_2 \not\perp\!\!\!\perp c_1$, then c_2 can be applied to another process only if c_1 is already applied.

Given the integrity properties, the representation and automated checking and inference of conflict and dependency relations [12, 37, 39, 67] is a topic of active research.

Transitions Rules. The transition rules of the operational semantics are presented in Figure 5. The rule Call accepts and executes an update method call c at the process p. It first checks that the call is locally permissible $\mathcal{P}(\sigma, c)$. It then checks that if the new call c conflicts with a call c' that another process p' has executed, then c' should have been already executed at the current process p. Thus, executing the call keeps the execution conflict-synchronizing. A replicated execution is conflict-synchronizing if every pair of conflicting calls have the same order across processes.

The rule Prop propagates a call c from another process p' to the current process p. Similar to the previous rule, this rule makes sure that executing c keeps the execution conflict-synchronizing. It checks that if a call c' that conflicts with the new call c is executed before c in any other process, then c' is already executed at the current process p. The rule also makes sure that executing c keeps the execution dependency-preserving. A replicated execution is dependency-preserving if for every call, its preceding dependencies in its issuing process precede it in the other processes as well. The rule checks that if a call c' is executed before c in p', and c is dependent on c', then c' is already executed at the current process p.

The rule Query executes a query call $q(v)$ at a process p. The return value v' is the result of applying the call to the current state σ of p.

We note that (op-based) CRDTs (Convergent and Commutative Replicated Data Types) [81] are a special case of WRDTs where it is assumed that all method calls state-commute with each other, and the integrity predicate is simply the assertion true. The conflict-synchronization and dependency-preservation conditions in the above transition rules are trivially satisfied. Therefore, the rules are always enabled and calls can propagate without coordination.

We also note that linearizable data types are a special case of WRDTs where the conflict relation is complete. The executions of WRDTs are conflict-synchronizing. Therefore, all the calls are totally ordered across processes. The execution histories $xs(p)$ of processes p are the prefixes of the total order. Further, the real-time preservation property is maintained by the Call rule as (1) a call c in process p returns only after adding c to $xs(p)$, and (2) the condition CallConfSync ensures that a process p' executes a call c' only after every call c that is in $xs(p)$ is also in $xs(p')$.

Guarantees. Every well-coordinated execution enjoys integrity and convergence.

Integrity is the safety property that the invariant predicate holds for all reachable states of a process.

Lemma 1 (Integrity). *For all ss and p, if* $W_0 \rightarrow^* \langle ss, \_ \rangle$ *then* $\mathcal{I}(ss(p))$.

Convergence is the safety property that states that processes which have applied the same set of calls have the same state. We say that two histories x and x' are equivalent $x \sim x'$ if they have the same set of calls.

Lemma 2 (Convergence). *For all ss, xs, p and p', if* $W_0 \rightarrow^* \langle ss, xs \rangle$ *and* $xs(p) \sim xs(p')$ *then* $ss(p) = ss(p')$.

$$\textsc{Call}\ \frac{c = u(v)_{p,r} \qquad \mathcal{P}(\sigma, c) \qquad \text{CallConfSync}(xs, p, c) \qquad xs' = xs\,[p \mapsto (xs(p) ::: c)] \qquad \sigma' = u(v)(\sigma)}{\langle ss[p \mapsto \sigma], xs\rangle \xrightarrow{p,\, u(v)_r} \langle ss[p \mapsto \sigma'], xs'\rangle}$$

$$\textsc{Prop}\ \frac{c = u(v)_{p',r} \qquad c \in xs(p') \setminus xs(p) \qquad \text{PropConfSync}(xs, p, c) \qquad \text{PropDepPres}(xs, p', p, c) \qquad xs' = xs\,[p \mapsto (xs(p) ::: c)] \qquad \sigma' = u(v)(\sigma)}{\langle ss[p \mapsto \sigma], xs\rangle \rightarrow \langle ss[p \mapsto \sigma'], xs'\rangle}$$

$$\textsc{Query}\ \frac{v' = q(v)(\sigma)}{\langle ss[p \mapsto \sigma], \_\rangle \xrightarrow{p,\, q(v):v'} \langle ss[p \mapsto \sigma], \_\rangle}$$

$$\text{CallConfSync}\,(xs, p, c) \;:=\; \forall c', p'.\; c' \in xs(p') \wedge c \bowtie c' \rightarrow c' \in xs(p)$$

$$\text{PropConfSync}\,(xs, p, c) \;:=\; \forall c', p'.\; c' \prec_{xs(p')} c \wedge c \bowtie c' \rightarrow c' \in xs(p)$$

$$\text{PropDepPres}\,(xs, p', p, c) \;:=\; \forall c'.\; c' \prec_{xs(p')} c \wedge c \not\perp c' \rightarrow c' \in xs(p)$$

Figure 5. WRDTs Semantics

3.3 RDMA Replicated Data Types

We now present the operational semantics of RDMA replicated data types. The semantics divides methods into three categories, reducible, irreducible conflict-free, and conflicting, and presents dedicated coordination requirements for each. We prove that this concrete semantics refines the abstract semantics of WRDTs that we saw in the previous subsection. This concrete semantics captures the core of our runtime system that we will see in section 4.

Method Categories. A pair of methods u and u' *conflict* if there are arguments v and v' such that the calls $u(v)$ and $u'(v')$ conflict. We say that a method is *conflicting* if there is a method that it conflicts with, and say that it is *conflict-free* otherwise. Similarly a method is *dependent* on another method if there is a call on the former that is dependent on a call on the latter. We write the set of methods that a method u is dependent on as $\text{Dep}(u)$. We say that a method is *dependence-free* if its set of dependencies is empty.

As we saw in the semantics of WRDTs, calls to a pair of conflicting methods should preserve the same order across processes. The conflict relation on methods induces an undirected graph that we call the conflict graph. The *synchronization* group $\text{SyncGroup}(u)$ of a method u is the connected component of the method in the conflict graph. Methods of a synchronization group synchronize with each other.

The summary of two calls c and c', written as $\text{Summarize}(c, c')$, is a call c'' iff for all states σ, $c \circ c'(\sigma) = c''(\sigma)$. For example, the summary of deposit(3) and deposit(4) is deposit(7). We say that a set of methods g are a *summarization group* if calls on g are closed under summarization. A sequence of calls from a summarization group can be successively summarized into a single call. We say that a method u is *summarizable* if it is a member of a summarization group, that we write as $\text{SumGroup}(u)$. Otherwise, it is not summarizable, that we write as $\text{SumGroup}(u) = \perp$. We say that a method is *reducible* if it is conflict-free, dependence-free and summarizable. Otherwise, we say that it is *irreducible*.

The semantics utilizes the remote write feature of RDMAs to directly communicate updates from one process to another. In order to tolerate faults and also have low latency for query methods, each process keeps a local replica of the state, and performs remote writes but no remote reads. The semantics distinguishes between three categories of methods: (1) conflicting methods, (2) irreducible conflict-free methods, *i.e.*, methods that are conflict-free but are either not dependence-free or not summarizable, and (3) reducible methods. We consider the coordination for each category in turn.

For conflicting methods, each synchronization group is assigned a leader process, and each process replicates a buffer of calls for each group. As we will see in the transition rules, the leader process of the group orders the calls on the group and then remotely appends them to the buffer of each other process. The other processes periodically traverse their own buffers and locally apply the calls. The leader is the single remote writer of all buffers, and each process is the single reader of its own local buffer.

Conflict-free methods do not need synchronization and processes autonomously issue and propagate them. Each process replicates a buffer of calls for all the irreducible conflict-free calls of each other process. When a process p issues an irreducible conflict-free call, it remotely appends it to the buffers that each other process stores for p. The other processes periodically traverse their buffer, locally apply the calls and then discard them. The process p is the single remote writer of these buffers, and each process is the single reader of its own local buffer.

Reducible methods are remarkable: the issuing process can reduce them together locally and then remotely write them for other processes (*i.e.*, using RDMA one-sided communication). Therefore, for each pair of a summarization group of methods and a process, each process replicates a single call rather than a buffer of calls. This not only saves space but also time as it eliminates the buffer traversals by the target processes. Thanks to direct RDMA writes, other processes obliviously receive updates without receiving and traversing messages. Similar to the above buffers, each summarization call is written by only a single remote process and is only read by the local process.

Replicated State. Figure 6 shows the state of the operational semantics. A configuration K is a mapping from processes p to tuples $\langle \sigma, A, S, F, L\rangle$. The stored state σ represents the result of applying calls at process p. These calls are either conflicting or irreducible conflict-free. The applied calls A is a mapping that maps a process p' and an update method u to the number of calls on u from p' that are applied

$$
\begin{array}{rcll}
g : G & = & \mathsf{Set}(U) & \text{Method Group} \\
A : \mathcal{A} & = & P \to U \to \mathsf{Nat} & \text{Applied calls} \\
D : \mathcal{D} & = & P \to U \to \mathsf{Nat} & \text{Dependencies} \\
S : \mathcal{S} & = & G \to P \to C & \text{Summarized calls} \\
F : \mathcal{F} & = & P \to \mathsf{List}\ (C \times \mathcal{D}) & \text{Conflict-free buffers} \\
L : \mathcal{L} & = & G \to \mathsf{List}\ (C \times \mathcal{D}) & \text{Conflicting buffers} \\
K : \mathcal{K} & = & P \to \Sigma \times \mathcal{A} \times \mathcal{S} \times \mathcal{F} \times \mathcal{L} & \text{Configuration}
\end{array}
$$

Figure 6. WRDT RDMA State

in the current process. The summarized calls S is a mapping that maps a summarization group g of update methods and a process p' to a call c that summarizes calls on methods in g from p'. The conflict-free calls F is a mapping that maps a process p' to the list of irreducible conflict-free calls received from p'; each call c is coupled with its dependencies D. The conflicting calls L is a mapping that maps a synchronization group g of update methods to the list of calls on methods of g; as before, each call c comes with its dependencies D.

Given a summarized map of calls S, the state $\mathsf{Apply}(S)(\sigma)$ is the result of applying the calls in the range of S to σ. Since the calls in the summarized map are conflict-free, they can be applied in any order. An applied map A satisfies a dependency map D, written as $D \leq A$, iff for all p and u, $D(p,u) \leq A(p,u)$.

We note that the semantics explicitly models the leader of each synchronization group that orders the calls on that group. The map L stores a copy of the total order at each process. The leader updates the orders stored at other processes by remote writes. Since these orders are the same, this model is a refinement of an abstract leaderless model where a configuration stores a single copy of the order.

Transition rules. The rule Reduce presents the transition for a reducible method call $u(v)$ by a process p_j. If u is conflict-free (*i.e.*, $\mathsf{SyncGroup}(u) = \bot$), is dependence-free (*i.e.*, $\mathsf{Dep}(u) = \emptyset$), and is summarizable (*i.e.*, $\mathsf{SumGroup}(u) = g$), then the call $u(v)$ can be reduced. The rule first checks that the call is locally permissible. The current state σ of this process p_j is calculated by applying the summarized calls that it has received S_j to its stored state σ_j. The post-state that results from applying $u(v)$ to σ should preserve the integrity property $\mathcal{I}$. The current summarizing call $u'(v')$ for the group g and the new call $u(v)$ are summarized as the call $u''(v'')$. The new summary call is stored at all the processes p_i, both locally at the current process p_j and remotely at other processes. Consequently, the number of applied calls on method u from p_j is incremented locally and stored both locally and remotely. The two remote writes are independent and can be issued concurrently.

The rule Free presents the transition for a conflict-free but irreducible (*i.e.*, dependent or not summarizable) method call $u(v)$ by a process p_j. If u is conflict-free (i.e, $\mathsf{SyncGroup}(u) = \bot$), but either dependent (*i.e.*, $\mathsf{Dep}(u) \neq \emptyset$) or not summarizable (*i.e.*, $\mathsf{SumGroup}(u) = \bot$), then the call $u(v)$ cannot be reduced but can avoid synchronization. As before, the call is first checked to be locally permissible. Then, the call is locally applied and the number of applied calls on u is incremented. It is then remotely written for each other process p_i: it is appended to the list $F_i(p_j)$ that stores at p_i the conflict-free calls issued from p_j. Let the dependencies $\mathsf{Dep}(u)$ of u be

Reduce

$$
\frac{\begin{array}{c}\mathsf{SyncGroup}(u) = \bot \qquad \mathsf{Dep}(u) = \emptyset \\ \mathsf{SumGroup}(u) = g \qquad \sigma = \mathsf{Apply}(S_j)(\sigma_j) \qquad \mathcal{I}(u(v)(\sigma)) \\ S_j(g,p_j) = u'(v') \qquad \mathsf{Summarize}(u'(v'),u(v)) = u''(v'') \\ \overline{S'_i = S_i[(g,p_i) \mapsto u''(v'')]}_{i \in \{1..|P|\}} \\ n = A_j(p_j,u)+1 \qquad \overline{A'_i = A_i[(p_i,u) \mapsto n]}_{i \in \{1..|P|\}}\end{array}}{\overline{[p_i \mapsto \sigma_i, A_i, S_i, \_, \_]}_{i \in \{1..|P|\}} \xrightarrow{p_j,\, u(v)} \overline{[p_i \mapsto \sigma_i, A'_i, S'_i, \_, \_]}_{i \in \{1..|P|\}}}
$$

Free

$$
\frac{\begin{array}{c}\mathsf{SyncGroup}(u) = \bot \qquad \mathsf{Dep}(u) \neq \emptyset \vee \mathsf{SumGroup}(u) = \bot \\ \sigma'_j = u(v)(\sigma_j) \qquad \sigma' = \mathsf{Apply}(S_j)(\sigma'_j) \qquad \mathcal{I}(\sigma') \\ A'_j = A_j[(p_j,u) \mapsto A_j(p_j,u)+1] \qquad \mathsf{Dep}(u) = \{\overline{u'}\} \\ \overline{F'_i = F_i[p_j \mapsto F_i(p_j) ::: \langle u(v), A_j|\{\overline{u'}\}\rangle]}_{i \in \{1..|P|\}\setminus\{j\}}\end{array}}{[p_j \mapsto \sigma_j, A_j, \_, \_, \_]\overline{[p_i \mapsto \_, \_, \_, F_i, \_]}_{i \in \{1..|P|\}\setminus\{j\}} \xrightarrow{p_j,\, u(v)} [p_j \mapsto \sigma'_j, A'_j, \_, \_, \_]\overline{[p_i \mapsto \_, \_, \_, F'_i, \_]}_{i \in \{1..|P|\}\setminus\{j\}}}
$$

Conf

$$
\frac{\begin{array}{c}\mathsf{SyncGroup}(u) = g \qquad \mathsf{Leader}(g) = p_j \\ \sigma'_j = u(v)(\sigma_j) \qquad \sigma' = \mathsf{Apply}(S_j)(\sigma'_j) \qquad \mathcal{I}(\sigma') \\ A'_j = A_j[(p_j,u) \mapsto A_j(p_j,u)+1] \qquad \mathsf{Dep}(u) = \{\overline{u'}\} \\ \overline{L'_i = L_i[g \mapsto L_i(g) ::: \langle u(v), A_j|\{\overline{u'}\}\rangle]}_{i \in \{1..|P|\}\setminus\{j\}}\end{array}}{[p_j \mapsto \sigma_j, A_j, \_, \_, \_]\overline{[p_i \mapsto \_, \_, \_, \_, L_i]}_{i \in \{1..|P|\}\setminus\{j\}} \xrightarrow{p_j,\, u(v)} [p_j \mapsto \sigma'_j, A'_j, \_, \_, \_]\overline{[p_i \mapsto \_, \_, \_, \_, L'_i]}_{i \in \{1..|P|\}\setminus\{j\}}}
$$

Free-App

$$
\frac{D \leq A \qquad \sigma' = u(v)(\sigma) \qquad A' = A[(p',u) \mapsto A(p',u)+1]}{K[p \mapsto \sigma, A, \_, F[p' \mapsto \langle u(v), D\rangle :: l], \_] \to K[p \mapsto \sigma', A', \_, F[p' \mapsto l], \_]}
$$

Conf-App

$$
\frac{\begin{array}{c}D \leq A \qquad \sigma' = u(v)(\sigma) \\ \mathsf{Leader}(g) = p' \qquad A' = A[(p',u) \mapsto A(p',u)+1]\end{array}}{K[p \mapsto \sigma, A, \_, \_, L[g \mapsto \langle u(v), D\rangle :: l]] \to K[p \mapsto \sigma', A', \_, \_, L[g \mapsto l]]}
$$

Query

$$
\frac{\sigma' = \mathsf{Apply}(S)(\sigma) \qquad v' = q(v)(\sigma')}{K[p \mapsto \sigma, \_, S, \_, \_] \xrightarrow{p,\, q(v):v'} K[p \mapsto \sigma, \_, S, \_, \_]}
$$

Figure 7. RDMA WRDTs Semantics

the set of methods $\{\overline{u'}\}$. The call $u(v)$ is accompanied with a record of applied calls that $u(v)$ is dependent on, *i.e.*, $A_j|\{\overline{u'}\}$.

The rule Conf presents the transition for a conflicting method call $u(v)$ by a process p_j. The process p_j is the leader for the method u. As before, the call is first checked to be locally permissible. Then, the call is locally applied and the number of applied calls is advanced. Let the synchronization group of u be g (*i.e.*, $\mathsf{SyncGroup}(u) = g$). The call is remotely written for each other process p_i: the call is appended to the list $L_i(g)$ that stores at p_i the calls from the synchronization group g. As before, the call is accompanied with a record of its dependencies. The function Leader uniquely maps each synchronization group to a process. As we will see in the next section, changing the leader of a synchronization group from one process to another preserves this uniqueness.

The rule Free-App presents an internal transition by a process p to apply a call from its conflict-free buffers F. A call from the buffer can be applied only if the record of the already applied calls A at p is ahead of the dependencies D of the call. The stored state σ is updated and the record of applied calls A is advanced. The rule Conf-App is similar except that it applies a call from the conflicting buffers L.

Finally, the rule Query presents the transition of a query $q(v)$ by a process p. The return value v' results from applying $q(v)$ to the current state σ', which is calculated by applying the summarized calls S to the stored state σ.

We will see an implementation of this semantics in the next section.

Correctness. The RDMA WRDT semantics (Figure 7) refines the WRDT semantics (Figure 5): any trace that is observed from the former can be observed from the latter.

Lemma 3 (Refinement). *For all K and τ, if $K_0 \xrightarrow{\tau} K$, then there exists W, such that $W_0 \xrightarrow{\tau} W$.*

The refinement relation and the proof are available in the supplemental material.

The immediate corollaries are that executions of RDMA WRDTs enjoy integrity and convergence.

All the reachable states of each process satisfy the integrity property. The state of a process is the result of applying the summarized calls S to the stored state σ.

Corollary 1 (Integrity). *For all $i \in \{1..|P|\}$, if $K_0 \rightarrow^* \overline{[p_i \mapsto \sigma_i, \_, S_i, \_, \_]}_{i\in\{1..|P|\}}$ then $\mathcal{I}(\mathsf{Apply}(S_i)(\sigma_i))$.*

When all the buffers F and L are applied, the states of the processes converge.

Corollary 2 (Convergence). *For all $i, j \in \{1..|P|\}$, if $K_0 \rightarrow^* \overline{[p_i \mapsto \sigma_i, \_, S_i, F_i, L_i]}_{i\in\{1..|P|\}}$ and $F_i = F_j = \emptyset$ and $L_i = L_j = \emptyset$ then $\mathsf{Apply}(S_i)(\sigma_i) = \mathsf{Apply}(S_j)(\sigma_j)$.*

4 Implementation

Hamband[1] is implemented on top of RDMA's Reliable Connection (RC) model using ibverbs library over Infiniband [1] in 1430 lines of code. First, we briefly explain the metadata stored at each node and then describe how Hamband propagates calls. In particular, we describe the reliable broadcast protocol that we use to broadcast conflict-free calls.

Meta-data. As we saw in the semantics, each node stores a location S for each reduction group, and two separate set of buffers: conflicting calls L, and irreducible conflict-free calls F. Each buffer has a head that is locally stored at the host node and a tail that is remotely stored at the single writer node. The buffers store pairs of calls c and their dependencies D. The dependency map that we saw in the semantics is efficiently represented as an array per node where each cell represents the number of calls on a method. Since the number of dependencies of methods is not necessarily the same, the dependency arrays are variable-sized. When the pairs of a buffer are traversed, the size of dependency arrays in the second element is decided based on the identifier of the method in the first element. Each node also keeps the number of applied calls A from each node as an integer array that is indexed by method identifiers. Further, each node keeps the following coordination analysis results: the list of synchronization groups, and a mapping from each method to the set of methods that it is dependent on.

Processing requests. Upon receiving a call request from the client, there are four possibilities based on the category of the method. First, if the call is a query, it is executed locally and the result is returned back to the client. Second, if the call is reducible, it is reduced with the local summary, and the result is remotely overwritten to the remote summary locations. Third, if the call is irreducible conflict-free, it is executed locally and written to the remote buffers F. To guarantee convergence, the above two propagations are done using the reliable broadcast abstraction. Before propagation, a call is assigned a unique id, paired with its dependency arrays and is serialized into a byte stream. Fourth, we instantiate a Mu [7] consensus instance for each synchronization group. If the call belongs to a synchronization group, it is sent to the corresponding consensus instance to be ordered in the L buffers.

In each node, two threads traverse and process the calls of F and L buffers if their dependencies are already satisfied. Each buffer has a head that is locally stored at the receiver node, and a tail that is remotely stored at the single writer node. Each call in the buffer contains a canary bit as the last bit. To check whether the buffer is not empty and the call is not concurrently being written, the receiver checks the canary bit. If the check fails, then the periodical traversal of the buffer will retry later. Even if a call is missed in a traversal, it will be processed in the next one. After a successful read,

[1] https://github.com/fhoushmand/Hamband.git

the head pointer is advanced to the next location. The calls at locations before the head are already executed. To avoid memory overflow, these locations are reused.

RDMA Reliable Broadcast. The reliable broadcast abstraction guarantees the following agreement property: if a message m is delivered by some correct node, then m is eventually delivered by every correct node. The best-effort broadcast abstraction on RDMAs can simply write the message remotely for all nodes. However, the source node may crash in the middle of the remote writes and violate the agreement property. To provide agreement, the source node keeps a shared memory location, and gives other nodes read access to the location. The source locally writes in the shared location before remotely writing it for others. It clears the location afterwards. The shared location acts as a backup. Each node has a heartbeat thread that periodically updates a local counter. This counter is periodically read by other nodes to determine whether that node is still alive or not. If other nodes detect that the source has failed, they remotely read the shared location, obtain any pending message, and check if they have received it. If not, they deliver the message.

Synchronization. We adopt Mu consensus protocol [7] to serialize calls in the L buffers. Under normal execution, only a designated leader has the permission to write to the follower buffers. As we described above, nodes have a heartbeat mechanism to let others detect when they fail. If a follower suspects that the leader has failed, it requests others to accept it as the leader and waits for a majority of them to acknowledge. At any time, each node recognizes only one node as the leader and grants it the write permission. A node revokes permission from the previous leader before granting it to the next. Therefore, only one node can be recognized as the leader by a majority and write to L buffers.

5 Experimental Results

We now evaluate the RDMA WRDTs of Hamband; we compare them with message-passing CRDTs and RDMA-enabled SMRs. We observe that Hamband outperforms message-passing CRDTs by 17.7× and 23× in terms of throughput and response time respectively. Further, it provides 2.7× higher throughput than the state-of-the-art SMR system, Mu, with almost the same response time.

Mu [7] is a low-latency leader-based SMR system, such as ZAB [40] that is used in the industry. However, we note that RDMA WRTDs are independent of any leader-based SMR protocol. They modularly use an SMR system (*i.e.*, consensus) for the conflicting category of methods. On the other hand, for the two conflict-free categories of methods, they avoid the synchronization cost and use more efficient broadcast protocols or just single RDMA writes. Therefore, they improve performance over the SMR baseline.

Questions. In our experiments, we aim to answer the following questions in terms of both throughput and response time: (1) How do RDMA CRDTs compare to message-passing CRDTs? (2) How do RDMA WRDTs compare to RDMA-based SMRs? We further investigate the following more detailed questions for RDMA WRDTs. (1) What is the effect of summarization and remote writing for reducible methods? (2) What is the effect of remote buffering for conflict-free methods? (3) What is the effect of separate synchronization groups for conflicting methods? (4) What is the impact of failures?

Platform and setup. We performed the experiments on a 7-node cluster, each with 8 AMD opteron 6376 cores and 50GB memory. The nodes are connected via 40Gbps Infiniband network, and run CentOS 7.4 Linux x86_64 kernel version 3.10. All programs are compiled with gcc-7.4.0.

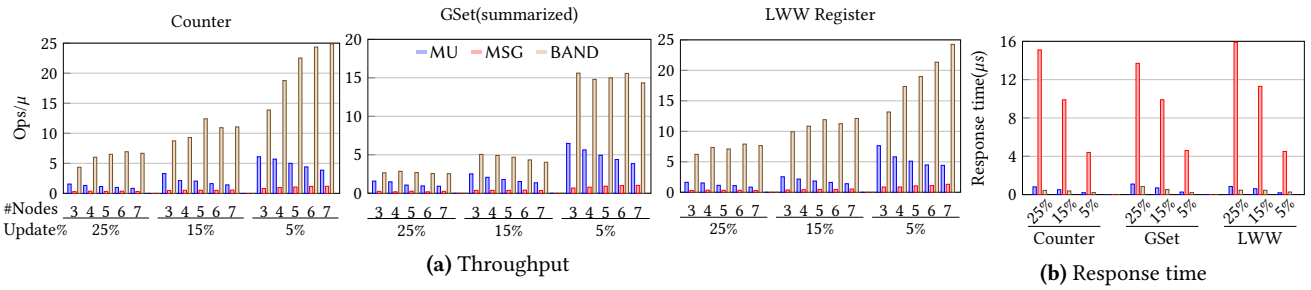
All the experiments are done with 4M operations unless stated otherwise. We randomly generate method calls and uniformly distribute update calls between updated methods. The calls on conflicting methods are automatically redirected to the corresponding leader node(s). All the other calls including conflict-free and query calls are divided equally between the nodes.

The throughput is calculated by dividing the total number of calls by the time that it takes for all the update calls to be replicated on all the nodes. The response time is calculated as the average response time over all the calls. We repeat each experiment 3 times and report the average.

Experiments and findings. We perform separate experiments for each of three categories of methods: reducible, irreducible conflict-free, and conflicting. We then experiment on the RDMA WRDT of a database schema that has all the three categories of methods. The results indicate that when there is no conflict, Hamband delivers on average 17.7× and 3.7× higher throughput than message-passing CRDTs and Mu SMR respectively. Even when there are conflicting calls, it delivers 1.7× higher throughput than Mu SMR. It improves the throughput of the database schema by up to 21% compared to the Mu SMR.

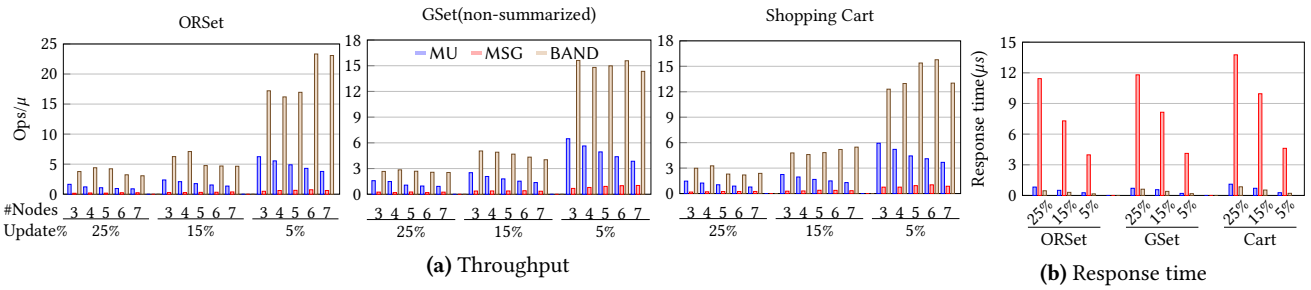
When there is no conflict, Hamband shows 23× lower average response time than message-passing CRDTs and almost the same response time for Mu SMR. When there are conflicting calls, the response time has an overhead of as little as 8% w.r.t Mu SMR. Further, we inject failures into the RDMA WRDT of a database schema. The experiment shows that it is able to tolerate leader or follower failure with minimum overhead for both throughput and response time of conflict-free operations.

Use-cases and benchmarks. We adopt [81] and experiment on the following five CRDTs: Counter, Last-writer-wins register (LWW), Grow-only set (GSet), Observed-Remove Set (ORSet), and Shopping cart. Moreover, we adopted [39, 71] three relational schemata: project management, courseware, and movie. The project management class has five methods, namely, addProject, deleteProject, worksOn, addEmployee, and query. The methods addProject, deleteProject, and worksOn belong to a synchronization group and the worksOn



(a) Throughput **(b)** Response time

Figure 8. Effect of summarization and remote writes for reducible methods



(a) Throughput **(b)** Response time

Figure 9. Effect of remote buffering for irreducible conflict-free methods

method depends on addProject and addEmployee due to the foreign-key constraint. The movie class has four methods addCustomer, deleteCustomer, addMovie, and deleteMovie operating on two separate relations; therefore, forming two synchronization groups. There is no dependency in this class. The Courseware class has five methods, namely, addCourse, deleteCourse, enroll, registerStudent, and query. Conflict analysis shows that there is one synchronization group that includes addCourse, deleteCourse and enroll. The enroll method depends on both addCourse and registerStudent.

Effect of reduction. RDMA WRDTs summarize and remotely write reducible calls. We study the effect of reduction on throughput and response time for three CRDTs with reducible operations: Counter, LWW and GSet. We consider three workloads that consist of 25, 15 and 5% update call ratios. Figure 8(a) and (b) compare the throughput and response time for Mu, message-passing CRDTs (MSG) and HAMBAND implementations. Figure 8(a) shows the scalability of HAMBAND's throughput. As the number of nodes increases or the ratio of updates decreases, the throughput of HAMBAND exhibits an increasing trend. HAMBAND delivers 18.4× and 4.1× higher throughput than MSG and Mu respectively. It delivers a throughput of up to 25 ops/μs. Figure 8(b) shows average response time for the three implementations on four nodes. (A larger graph is available in the appendix.) We observe that the response time of HAMBAND is on average 21× lower than MSG and almost the same as Mu. Decreasing the update ratios results in lower response times across the board. Mu serializes the calls. The MSG implementation avoids synchronization and simply sends messages. In addition to synchronization avoidance, HAMBAND benefits from summarization and remote writes. Therefore, it exhibits higher throughput and lower response time than MSG and Mu.

Previous work has shown the benefits of both one-sided [32, 63] and two-sides [45] communication in different contexts. Our replicated data types can benefit from one-sided verbs to perform one of our three method categories in one shot and improve performance.

Effect of remote buffering. There are methods that are simply not reducible, either because they have conflicts or dependencies, or because they are not summarizable. Figure 9(a) and (b) compare the throughput and response time of Mu, MSG and HAMBAND for three CRDTs with irreducible conflict-free operations: ORSet, GSet and Shopping Cart. (The methods of GSet are reducible; however, here, we use an implementation that uses buffers instead of summaries.) HAMBAND exhibits 17× and 3× higher throughput than MSG and Mu respectively. It delivers a throughput of up to 23 ops/μs. Similarly, we observe an average of 24.3× lower response times compared to MSG and almost the same response times as Mu. Similar to the previous experiment for reducible methods, this experiment shows that coordination avoidance and direct accesses for propagation lead to higher throughput and lower response time for irreducible conflict-free methods. However, the gains for reducible methods were higher since they do not need remote iteration and application of the buffered calls.

Effect of synchronization groups. To study the effect of synchronization groups, we compared HAMBAND and Mu

on the movie use-case whose methods form two distinct synchronization groups. We perform experiments that execute 2, 4, and 8M update operations on four nodes. Figure 10(a) and (b) show the throughput and response time respectively. We observe that the Hamband exhibits 1.4× to 1.8× higher throughput than Mu. This is due to the fact that Hamband is able to utilize two separate leaders to order requests while Mu uses a single leader. Hamband's throughput gain is close to the theoretical limit of 2×. The difference in the response times is statistically negligible because the synchronization operations that a leader performs for a call is independent of the number of leaders.

Mix of categories. In this subsection we experiment with the project management database scheme that has methods in all the three categories. Figure 11(a) compares the throughput of Hamband and Mu with 50%, 25%, and 10% update calls on four nodes. Hamband provides up to 21% higher throughput than Mu. Figure 11(b) compares the response times for each method. The response times for all methods except WorksOn stay almost the same. The response time for WorksOn calls is higher since they are dependent on addProject and addEmployee calls and have to wait for them to be delivered.

Fault tolerance. We now study the effect of failure on throughput and response time. All the failure experiments are done on 4 nodes. We first experiment on two CRDTs in Figure 12 to study the effect of failure where there is no conflicting method. The methods of these use-cases are all in the two conflict-free categories. Therefore they use the reliable broadcast protocol or the single RDMA writes and do not use Mu. Moreover, we report results for the more elaborate courseware WRDT that has methods in all the three categories in Figure 13. We inject failures into a node by suspending its heartbeat thread which make other nodes suspect that node. After a failure, all the requests of the failed node are redirected to the next available node. In the case of leader failure, the conflicting calls have to wait until the leader-change protocol elects the new leader.

Figure 12(a) and (b) show the throughput and the response time of the Counter and ORSet respectively with different update ratios. We observe that the throughput of the Counter and ORSet decrease by only an average of 5% and 5%, while the average response time increases by 15% and 5% respectively. Therefore, Hamband can smoothly withstand failures for conflict-free use-cases.

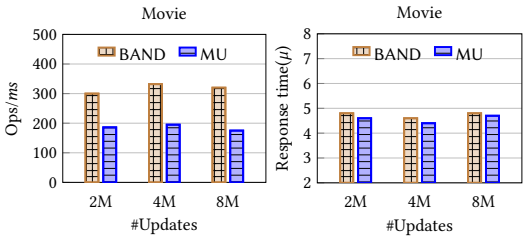
Figure 13(a) and (b) show the throughput and average response time of the courseware use-case for three scenarios: the normal execution without failures as the baseline, failure of a follower, and finally, failure of the leader. Figure 13(a) shows that Hamband can gracefully tolerate follower failure with only 6% impact on the throughput. However, since the leader change protocol is involved, when the leader of the synchronization group fails, the decrease in throughput is 53%. Figure 13(b) shows the response time per method. The response time of the conflict-free registerStudent method experiences little to no change even in the leader failure scenario. This is because calls on this method do not need to be synchronized by the leader; therefore, they can be easily redirected to follower nodes without much impact on their response time. However, the response time of the conflicting methods such as addCourse, deleteCourse, and enroll almost doubles when the leader fails. They need to wait for the leader-change protocol to install the next leader.

6 Related Works

RDMA and hardware-aided replication. A few replication systems have been recently designed for RDMA [7, 41, 48, 74, 86] but they all implement an SMR and provide strong consistency. In contrast, this paper considers the semantics of methods, and avoids synchronization when possible.

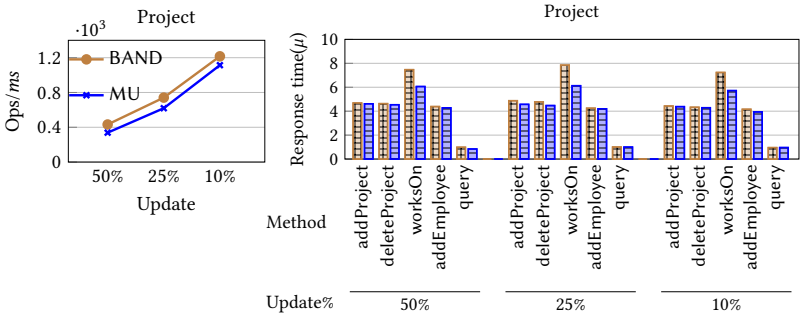
DARE [74], the first RDMA-based SMR, presented a wait-free protocol that uses RDMA direct accesses and permissions, and applies it to implement a strongly consistent key-value store. Subsequently, APUS [86] improved the throughput of the SMR protocol. However, it showed higher response times, since the protocol requires the followers assist the leader during replication. Derecho [41], supports both an in-memory and a persistent SMR with high throughput. It uses an RDMA multicast protocol (RDMC) to move the data in high-rate flows, and uses a distributed shared memory (SST) to exchange control messages that determine when it is safe to deliver the data. Mu [7] reaches consensus with a single one-sided RDMA operation in the common case. It uses remote reads to detect failures and uses permissions to prevent concurrent leaders in the case of failure. Our synchronization mechanism for conflicting methods is similar. Hermes [48] uses logical timestamps to decentralize write operations, and locally establish a total order for a key-value store. Therefore, it is similar to the last-write-wins register CRDT that we implemented as well. In contrast, this paper offers general semantics and protocols for WRDTs that subsume CRDTs. Odyssey [34] presents a taxonomy and a comparison of these replicated systems.

Kite [35] adopts the release consistency (RC) model from shared-memory concurrency where threads use release and acquire synchronization primitives, and offers these primitives in a high throughput key-value store abstraction (similar to a distributed shared memory). The key value store is implemented on top of eventually and strongly consistent protocols that benefit from RDMA acceleration, and provides the well-understood SC for DRF guarantee. On the other hand, Hamband takes a high-level data type with no distribution details together with integrity properties. The convergence and integrity requirements lead to the inference of conflict and dependency relations between methods. According to these relations, Hamband categorizes methods



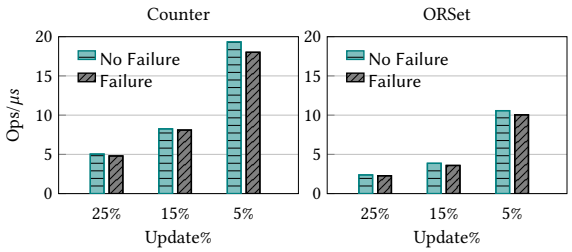
(a) Throughput **(b)** Response time

Figure 10. The effect of synchronization groups

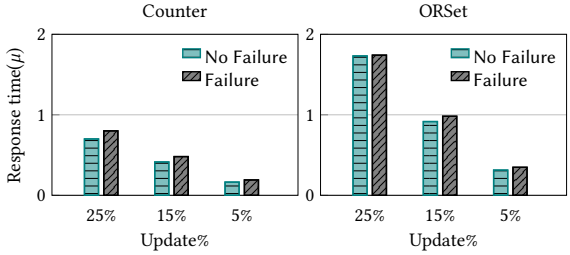


(a) Throughput **(b)** Response time

Figure 11. Project management use-case

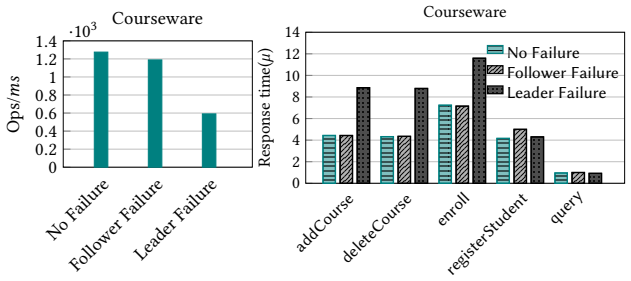


(a) Throughput



(b) Response time

Figure 12. Effect of failure on the Courseware and ORSet use-cases.



(a) Throughput **(b)** Response time

Figure 13. The effect of failure on the courseware use-case.

into three classes based on their coordination requirements that are separately and efficiently implemented on top of reliable and total-order broadcast protocols on RDMA. In particular, conflict-free methods calls can be executed under eventual instead of sequential consistency. Further, the reducible class of method calls can be implemented as single RDMA remote writes.

NetChain [42] uses programmable switches to store data and process queries in the network data plane. This eliminates the query processing at coordination servers and reduces the response time. HovercRaft [49] extends the Raft protocol to separate request replication from ordering, and integrates it with a transport protocol on a P4 [31] ASIC that supports load-balancing by updating the destination IP of RPC requests.

Hybrid replication models. Several projects have recently considered hybrid consistency models. However, all of them assumed the traditional message-passing network model; none addressed replication on the RDMA network model and its one-sided communication mechanism.

IPA [11] presents a static analysis the identifies the conflicting operations that can violate the integrity properties, and modifies them such that the invariants are maintained. Sieve [55–57] applies static and dynamic analysis to determine whether an operation can be executed under causal consistency (blue class) or needs strong consistency (red class) in order to preserve the invariants. Quelea [83] and similarly the follow-up works [16, 26] define axiomatic semantics for consistency notions based on primitive consistency relations such as visibility and session orders. They capture user-defined consistency contracts for methods using the same primitives. They then automatically map a contract to the weakest consistency notion that satisfies the contract. Indigo [12, 13] captures invariants and post-conditions of methods in terms of user-defined predicates. It then identifies conflicting methods and either prevents or repairs their concurrent executions. CISE [37, 68] allows the user associate tags with methods and define conflicts between tags, and

presents a rely-guarantee style proof technique for invariant preservation. Hamsaz [39] presents an axiomatic definition of well-coordination; in contrast, this paper presents an abstract operational semantics for general WRDTs, and further a concrete operational semantics for RDMA WRDTs, and proves a refinement between them. Carol [54] lets users declare required guard predicates on the current and remote view of the data, and automatically infers the required coordination. In order to reduce coordination, ECRO [30] reorders conflicting operations locally when possible.

7 Conclusion

We saw well-coordinated replicated data types (WRDTs) for the RDMA network model. We saw operational semantics for both abstract WRDTs and concrete RDMA WRDTs. The abstract semantics captures the well-coordination conditions and serves as a specification for the concrete semantics. The concrete semantics of RDMA WRDTs divides methods into three categories based on their conflict, dependency and summarization properties, and captures their coordination requirements based on one-sided communication. It is formally proved that the concrete semantics refines the abstract semantics and preserves convergence and integrity. We saw the protocols that efficiently implement the semantics, and the empirical evaluation that shows their high throughput.

We hope that this project motivates research for analysis and synthesis of distributed programs for the new RDMA network model.

References

[1] [n.d.]. InfiniBand Userspace verbs access. https://www.kernel.org/doc/html/latest/infiniband/user_verbs.html.

[2] [n.d.]. Mellanox Technologies. RDMA aware networks programming user manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[3] [n.d.]. Memcached. http://memcached.org/.

[4] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design. *Computer* 45, 2 (2012), 6 pages.

[5] Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. 2018. Passing messages while sharing memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. 51–60.

[6] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. 2019. The impact of RDMA on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 409–418.

[7] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond consensus for microsecond applications. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 599–616.

[8] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency analysis in Bloom: A CALM and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*. 249–260.

[9] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196.

[10] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2015. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1327–1342.

[11] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. [n.d.]. IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases. *Proceedings of the VLDB Endowment* 12, 4 ([n. d.]).

[12] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. ACM, New York, NY, USA, Article 6, 16 pages.

[13] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. 2015. Towards Fast Invariant Preservation in Geo-replicated Systems. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 121–125. https://doi.org/10.1145/2723872.2723889

[14] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[15] Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2021. Checking Robustness Between Weak Transactional Consistency Models. *Programming Languages and Systems* 12648 (2021), 87.

[16] Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 59. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[17] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: effectively testing correctness under weak isolation levels. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[18] A. Bouajjani, C. Enea, and J. Hamza. 2014. Verifying Eventual Consistency of Optimistic Replication Systems. In *Proc. POPL*.

[19] Eric Brewer. 2012. CAP twelve years later: How the" rules" have changed. *Computer* 45, 2 (2012), 23–29.

[20] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7. Portland, OR, 343477–343502.

[21] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 458–472.

[22] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 691–707.

[23] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. 2012. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*. Springer, 283–307.

[24] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proc. POPL*.

[25] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 335–350.

[26] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[27] Kevin Clancy and Heather Miller. 2017. Monotonicity Types for Distributed Dataflow. In *Proceedings of the Programming Models and Languages for Distributed Computing*. ACM, 2.

[28] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel

Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.

[29] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages.

[30] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: building global scale systems from sequential code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[31] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: formal foundations for p4 data planes. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32.

[32] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[33] Michael Emmi and Constantin Enea. 2018. Monitoring Weak Consistency. In *Proc. CAV*.

[34] Vasilis Gavrielatos, Antonios Katsarakis, and Vijay Nagarajan. 2021. Odyssey: The impact of modern hardware on strongly-consistent replication protocols. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 245–260.

[35] Vasilis Gavrielatos, Antonios Katsarakis, Vijay Nagarajan, Boris Grot, and Arpit Joshi. 2020. Kite: Efficient and available release consistency for the datacenter. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–16.

[36] Victor BF Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[37] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'M Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

[38] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. ACM, New York, NY, USA, 279–293.

[39] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. In *Proceedings of Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*. ACM, New York, NY, USA.

[40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11.

[41] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P Birman. 2019. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)* 36, 2 (2019), 1–49.

[42] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 35–49.

[43] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe replication through bounded concurrency verification. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[44] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[45] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

[46] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*. 437–450.

[47] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Fasst: Fast, scalable and simple distributed transactions with two-sided ({RDMA}) datagram rpcs. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 185–201.

[48] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 201–217.

[49] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–17.

[50] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. 2020. Rethinking safe consistency in distributed object-oriented programming. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[51] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[52] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998).

[53] Leslie Lamport. 2004. Generalized Consensus and Paxos. (2004).

[54] Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černỳ. 2019. Sequential programming for replicated data stores. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28.

[55] Cheng Li, João Leitão, Allen Clement, Nuno Preguica, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292.

[56] Cheng Li, João Leitão, Allen Clement, Nuno Preguica, and Rodrigo Rodrigues. 2015. Minimizing coordination in replicated systems. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, 8.

[57] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, and Rodrigo Rodrigues. 2012. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

[58] Xiao Li, Farzin Houshmand, and Mohsen Lesani. 2020. Hampa: Solver-Aided Recency-Aware Replication. In *International Conference on Computer Aided Verification*. Springer, 324–349.

[59] Hongjin Liang and Xinyu Feng. 2021. Abstraction for conflict-free replicated data types. In *Proceedings of the 42nd ACM SIGPLAN International*

Conference on Programming Language Design and Implementation. 636–650.

[60] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying replicated data types with typeclass refinements in Liquid Haskell. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[61] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. 184–195.

[62] Matthew Milano and Andrew C Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. (2018).

[63] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using {One-Sided}{RDMA} Reads to Build a Fast,{CPU-Efficient}{Key-Value} Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 103–114.

[64] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. 2019. A fault-tolerant programming model for distributed interactive applications. *Proceedings of the ACM on Programming Languages* OOPSLA (2019), 1–29.

[65] Kartik Nagar and Suresh Jagannathan. 2019. Automated parameterized verification of crdts. In *International Conference on Computer Aided Verification*. Springer, 459–477.

[66] Kartik Nagar, Prasita Mukherjee, and Suresh Jagannathan. 2020. Semantics, Specification, and Bounded Verification of Concurrent Libraries in Replicated Systems. In *International Conference on Computer Aided Verification*. Springer, 251–274.

[67] Sreeja Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the safety of highly-available distributed objects. In *ESOP 2020-29th European Symposium on Programming*.

[68] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE Tool: Proving Weakly-consistent Applications Correct. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data* (London, United Kingdom) *(PaPoC '16)*. ACM, New York, NY, USA, Article 2, 3 pages.

[69] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. ACM, New York, NY, USA, 8–17.

[70] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320.

[71] M Tamer Özsu and Patrick Valduriez. 2020. *Principles of distributed database systems*. Vol. 2. Springer.

[72] Seo Jin Park and John Ousterhout. 2019. Exploiting commutativity for practical fast replication. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 47–64.

[73] Fernando Pedone and André Schiper. 2002. Handling message semantics with generic broadcast protocols. *Distributed Computing* 15, 2 (2002), 97–107.

[74] Marius Poke and Torsten Hoefler. 2015. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 107–118.

[75] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: directed test generation for weakly consistent database systems. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

[76] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2021. Repairing serializability bugs in distributed database programs via automated schema refactoring. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 32–47.

[77] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354–368.

[78] Signe Rüsch, Ines Messadi, and Rüdiger Kapitza. 2018. Towards low-latency byzantine agreement protocols using RDMA. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 146–151.

[79] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[80] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. 2016. *Consistency in 3D*. Ph.D. Dissertation. Institut National de la Recherche en Informatique et Automatique (Inria).

[81] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report RR-7506. INRIA.

[82] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[83] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 413–424.

[84] Werner Vogels. 2008. Eventually consistent. *ACM Queue* 6, 6 (2008).

[85] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 980–993.

[86] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*. 94–107.

[87] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled distributed transactions: Hybrid is better!. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 233–251.

[88] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 87–104.

[89] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. 2020. Composing and decomposing op-based CRDTs with semidirect products. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–27.

[90] Michael Whittaker and Joseph M Hellerstein. 2018. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment* 12, 1 (2018), 14–27.

[91] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal specification and verification of crdts. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 33–48.