FRASHOKERETI: Non-aborting Optimistically Replicated Objects

ERIC MAN CHAN, University of California at Riverside, USA
JAVAD SABERLATIBARI, University of California at Riverside, USA
MOHSEN LESANI, University of California at Santa Cruz, USA

Optimistic replication of objects avoids coordination and brings higher responsiveness and availability. However, when clients issue concurrent operations, conflicts naturally arise which can lead the replicated states to diverge or lose integrity. When conflicts occur, existing approaches resort to pessimism or abortion. This paper characterizes ORDTs (Optimistically Replicated Data Types), objects that can be optimistically replicated with convergence and integrity, and without aborting calls. It shows that ORDTs subsume CRDTs and transformed relational schema, and presents techniques to convert objects to ORDTs. It further proves that optimistic replication for objects that fall out of ORDTs is aborting and NP-Complete. Further, it presents an optimistic replication protocol for ORDTs called FRASHOKERETI. It uses a statically decided order to efficiently order calls. The paper proves that FRASHOKERETI is sound for every ORDT, *i.e.*, FRASHOKERETI is optimistic and non-aborting, and preserves convergence, integrity, and liveness properties. Experimental results show that FRASHOKERETI significantly outperforms previous optimistic protocols.

CCS Concepts: • **Computing methodologies → Distributed algorithms**; • **Information systems →** *Distributed database transactions.*

Additional Key Words and Phrases: Data replication, optimistic replication

ACM Reference Format:
Eric Man Chan, Javad Saberlatibari, and Mohsen Lesani. 2026. FRASHOKERETI: Non-aborting Optimistically Replicated Objects. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 161 (April 2026), 29 pages. https://doi.org/10.1145/3798269

1 Introduction

Background. Distributed replication of objects brings multiple desirable properties including availability and fault-tolerance. The well-known classical technique for replication is state-machine replication (or total-order broadcast), which has the same computational power as consensus. Systems use this abstraction to provide strong consistency [22, 38, 50, 52]. Replicas pessimistically synchronize every operation to maintain the same order of operations across replicas. Thus, replicas preserve convergence and integrity (*e.g.*, uniqueness in a list of elements). However, repeated synchronization results in limited scalability, throughput and responsiveness, and is susceptible to network partitions [1, 18, 19, 27, 28].

To yield higher performance, a replicated system can be *optimistic*. Optimistic replication *avoids synchronization* (*i.e.*, expensive coordination). Optimistic replication protocols try to tentatively execute requested operations at the local replica [59]. The operation is then propagated throughout the system when possible. When replicas issue concurrent requests, conflicts naturally arise:

Authors' Contact Information: Eric Man Chan, University of California at Riverside, Riverside, USA, eric.chan008@email.ucr.edu; Javad Saberlatibari, University of California at Riverside, Riverside, USA, javad.saberlatibari@email.ucr.edu; Mohsen Lesani, University of California at Santa Cruz, Santa Cruz, USA, m.lesani@ucsc.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/4-ART161
https://doi.org/10.1145/3798269

operations can cross each other and make the replicated state diverge or lose integrity, affecting the client [66]. Thus, optimistic replication techniques have to resolve these conflicts to preserve convergence and integrity.

Related Works. A prominent class of optimistic replication is Conflict-free Replicated Data Types (CRDTs) [60, 61] (and similar notions [4, 5, 57]), which formally characterize a class of objects that always converge without ever coordinating. An example is the grow-only set which only allows adding elements. However, CRDTs require every pair of calls on the object to be commutative, limiting their applicability. Further, they are concerned with convergence, but not integrity [7]. On the other hand, a few works capture the conditions under which replicated data types can preserve integrity without coordination [6, 49].

However, convergence and integrity of many common objects cannot be preserved without synchronization, and synchronization dominates performance. Thus, so-called "hybrid" solutions such as IPA [8], Sieve [40–42], Indigo [9, 10], CISE [29], Quelea [62], Carol [39] Hamsaz [32, 43], and Q9 [33] perform synchronization judiciously. They categorize operations based on whether they may violate convergence or integrity without synchronization. They accordingly decide whether to execute an operation with synchronization. However, they perform pessimistic coordination: an operation is synchronized if it can likely (even if not certainly) violate integrity. For example, consider the referential integrity of a database schema. Adding a foreign key and concurrently deleting the primary key it refers to would violate integrity, thus add and delete operations must synchronize. Since adds and deletes are generally abundant, the cost of synchronization accrues, hindering performance.

Optimism can accelerate performance, but how can we preserve both convergence and integrity? Bayou [65] attaches preconditions to each operation, and a resolver for when the preconditions are not met; however, they have been shown to be highly application-specific [64]. ECRO [23] presents optimistically replicated objects that preserve convergence and integrity, although for certain invariants, it resorts to synchronization and locking. It performs operations optimistically and later reorders or aborts them if needed. ECRO invited further research by pointing out the abortion issue: "discarding the operation ... may cause an anomaly observed by a client. Future work could explore alternative ways ...". Aborting calls can affect clients. Further, ECRO keeps a runtime graph of calls and their relationships, resorting to solving non-polynomial graph problems to abort some of the calls, and order the rest for re-execution.

This Paper. The facts that ECRO is an *aborting* protocol and has *non-polynomial* local complexity pose the following two questions about optimistic protocols that replicate general objects, and preserve convergence and integrity. Are there non-aborting protocols? Are there protocols with polynomial-time local complexity? This paper responds negatively to both. We present example objects and executions that warrant abortion. Further, we show that optimistic replication for general objects is NP-Complete. We remember that CRDTs are non-aborting but preserve only convergence for a limited set of objects. This poses the following questions about optimistic protocols that preserve convergence and integrity. Which objects can have *non-aborting* protocols? Can we design such protocols with *polynomial-time* local complexity? This paper responds to both. It characterizes a superset of CRDTs called ORDTs (Optimistically Replicated Data Types). It shows that ORDTs subsume CRDTs and transformed relational schema, and presents abstraction and transformation techniques to convert objects to ORDTs. Further, the paper presents a protocol called Frashokereti[1] for ORDTs. We prove that Frashokereti is optimistic and non-aborting, and preserves convergence, integrity, and liveness properties. It avoids maintaining a call graph at

[1] Frashokereti is the Zoroastrian worldview where despite current challenges, the world will eventually return to its perfect state.

runtime, and has polynomial-time local complexity. Further, we implemented FRASHOKERETI, and empirically evaluated it on several non-conflicting and conflicting objects. Experimental results show that FRASHOKERETI significantly outperforms ECRO and scales. Further, it exhibits comparable performance to CRDTs for conflict-free objects.

We organize this paper as follows. In § 2, we present an overview. In § 3, we present the replication model, objects, histories, protocol specifications, and conflict relations on methods. In § 4, we define the ORDT class of objects, and present intuitions and observations to be used by the protocol. In § 5, we present the FRASHOKERETI protocol and in § 6 prove its correctness. In § 7, we show that ORDTs subsumes CRDTs and transformed relational schema. Further, we show how abstraction and transformation can yield acyclic conflict graphs. In § 8, we prove the NP-Completeness of optimistic protocols for general objects. In § 9, we present the experimental results.

2 Overview

In this section, we introduce ORDTs (Optimistically Replicated Data Types) with a relational schema example. We show how a static order of its methods can be defined, and further present example executions and an overview of the optimistic protocol FRASHOKERETI.

Conflicting calls can violate convergence and integrity. In order to preserve these properties, processes must agree on the execution order of concurrent conflicting calls. Pessimistic replication protocols either coordinate to order them, or temporarily lock each other from executing them. On the other hand, optimistic protocols allow processes to avoid coordination and make local decisions. However, in order to resolve conflicts, optimistic protocols sometimes resort to aborting calls that have already been executed, which is generally undesirable for users. The most common strategy is to combine pessimism and optimism into a hybrid approach: be optimistic and avoid coordination when possible, and otherwise be pessimistic and coordinate. In this work, we are interested in non-aborting, optimistically replicated objects: objects for which processes can always optimistically execute calls, never abort any call, and still preserve convergence and integrity.

We characterize these objects (in § 4) and present the optimistic protocol FRASHOKERETI for them (in § 5). The idea is to statically find a partial order over the object methods and use it at runtime. Processes then (locally) consult the partial order to order concurrent conflicting calls, rather than coordinating with each other. The partial order is chosen in such a way that when processes execute concurrent calls accordingly, no conflicts occur, thereby preserving convergence and integrity. Respecting the partial order entails subtle treatment of calls including the order of tentative calls, and waiting for stability of certain calls before executing others.

Project Management Relational Schema. We build the intuition with a project management database schema. The database state is defined by three sets: EMPLOYEES, PROJECTS, and WORKS, where EMPLOYEES is the set of employees, PROJECTS is the set of projects, and WORKS is the set of employee-project pairs representing which employees are working on which projects. The invariant $\mathcal{I}$ is the referential integrity from WORKS to both EMPLOYEES and PROJECTS: every employee/project identifier from a tuple in WORKS must refer to an existing employee/project in EMPLOYEES/PROJECTS. More formally, $\forall (e, p) \in$ WORKS. $e \in$ EMPLOYEES $\wedge$ $p \in$ PROJECTS.

The object has five methods: add-employee, add-project, delete-employee, delete-project, and works-on. The add-employee(e) and add-project(p) methods add e and p to EMPLOYEES and PROJECTS, respectively. The delete-employee(e) method removes e from EMPLOYEES, and cascades delete to every tuple in WORKS that refers to e. Similarly, delete-project(p) deletes p from PROJECTS, and every tuple in WORKS that refers to p. Lastly, the works-on(e, p) method adds the pair (e, p) to WORKS. All methods are idempotent. We implicitly assume every object in this paper has a query (read) method that accesses, but does not mutate the state.

Ordering State Conflicts. Consider two concurrent calls on add-project(p) and delete-project(p). These two calls state-conflict: processes must agree on their order to determine whether $p \in$ PROJECTS at the end. As Fig. 1 (top, undirected red edges) shows, state conflicts between pairs of methods can be represented as an undirected graph $\mathcal{G}_S$. We can statically order add-project before delete-project within our partial order. This denotes that when a process receives concurrent calls on these methods, calls on the former should be executed before calls on the latter (*i.e.*, the "delete-wins" semantics). Processes locally order the calls so that add-project(p) is first and delete-project(p) is second, resulting in $p \notin$ PROJECTS. We note that the partial order can pick the opposite ordering of the methods as well, hence the undirected edge.

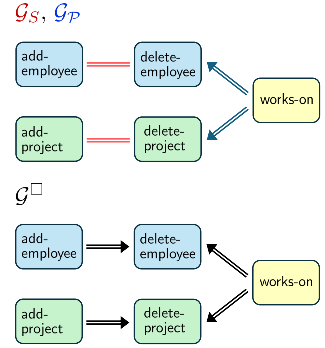

Fig. 1. Top: the state graph $\mathcal{G}_S$ and permissibility graph $\mathcal{G}_\mathcal{P}$ for the employee-project schema: $\mathcal{G}_S$ is induced by the undirected (red) edges and $\mathcal{G}_\mathcal{P}$ is induced by the directed (blue) edges. Bottom: one possible conflict graph $\mathcal{G}^\square$.

Ordering Permissibility Conflicts. Consider two concurrent calls on works-on$(Alice, p)$ and delete-project(p), and the two orders they can be executed in. If project p is first deleted, and Alice is assigned to work on p after, then integrity is violated. We say that the call works-on$(Alice, p)$ $\mathcal{P}_R$-conflicts (permissible-right conflicts) with delete-project(p). As Fig. 1 (top, directed blue edges) shows, permissible conflicts can be represented as a directed graph $\mathcal{G}_\mathcal{P}$. On the other hand though, if Alice is assigned p first, and p is deleted second, then both calls are permissible. We say the call delete-project(p) $\mathcal{P}_R$-commutes with works-on$(Alice, p)$. The partial order can also be used to prevent permissibility conflicts. The key observation is to respect the $\mathcal{P}_R$-commutativity of calls. Unlike state-commutativity, the "direction" matters, hence the directed edge; the delete-project method $\mathcal{P}_R$-commutes with the works-on method, but not vice-versa. Therefore, works-on should appear before delete-project in our partial order.

Partial Order and ORDTs. Generalizing the two examples above, we want to statically define a partial order over all the methods. For two methods m and m', if m and m' state-conflict, then they should be ordered, though it does not matter which precedes the other. On the other hand, if m $\mathcal{P}_R$-conflicts with m', then m should be ordered *before* m'. In order to define the partial order, we aggregate $\mathcal{G}_S$ and $\mathcal{G}_\mathcal{P}$ into a single, directed *conflict graph* $\mathcal{G}^\square$ (where $\square$ denotes static). Given $\mathcal{G}_S$ and $\mathcal{G}_\mathcal{P}$ of an object, $\mathcal{G}^\square$ must satisfy the following properties. The vertex set of $\mathcal{G}^\square$ is the same as $\mathcal{G}_S$ and $\mathcal{G}_\mathcal{P}$. Every (directed) edge in $\mathcal{G}_\mathcal{P}$ is in $\mathcal{G}^\square$. At least one directed variant of every (undirected) edge in $\mathcal{G}_S$ is in $\mathcal{G}^\square$. Thus, an object can have multiple conflict graphs. Fig. 1 (bottom) shows one such conflict graph $\mathcal{G}^\square$ for our running example. If a conflict graph $\mathcal{G}^\square$ is acyclic and loop-free, reachability induces a (strict) partial order, $\prec_\mathcal{R}$. We consider a new class of objects: an Optimistically Replicated Data Type (ORDT) is an object with an acyclic and loop-free conflict graph.

Intuitively, ORDTs enjoy the optimistic features of CRDTs while accommodating many application-specific integrity properties. As a formal comparison, an object is a CRDT only if its conflict graph is empty. In turn, many data structures and applications that are not CRDTs, such as (sequential) sets and stacks, are indeed ORDTs. Further, readers may most be interested to find that general relational database schema can be transformed into ORDTs, which we discuss in detail in § 7.3.

Replication Protocol. This paper presents a non-aborting, optimistic protocol for ORDTs. We give the intuition for our protocol here, and formally present it in § 5. A process executes a sequence of calls, which we refer to as the *log* of the process. The idea is for the log to mimic a (repeating) sequence of *buckets* along $\prec_\mathcal{R}$, where each bucket corresponds to a specific method. When a process receives a remote call, it inserts the call into its corresponding bucket in the tentative suffix of the log. By design, the buckets are arranged so that methods of later buckets are permissible after

methods of earlier buckets. That is, calls on methods later in $\prec_{\mathcal{R}}$ are guaranteed to $\mathcal{P}_R$-commute with all calls on methods earlier in $\prec_{\mathcal{R}}$, preserving the integrity of each call. Additionally, since every process sequences their buckets with the same partial order, convergence is preserved.

For example, consider the execution in Fig. 2. Processes p_1 and p_2 concurrently execute add-project(q_1) and add-employee$(Alice)$, respectively, and propagate these calls. Then, p_1 executes works-on$(Alice, q_1)$, while p_2 concurrently executes add-employee(Bob) and delete-project(q_1). When p_2 receives works-on$(Alice, q_1)$, since works-on $\prec_{\mathcal{R}}$ delete-project, p_2 inserts works-on$(Alice, q_1)$ before delete-project(q_1), preserving integrity.

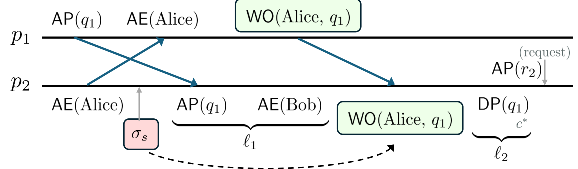

Fig. 2. An example execution on our employee-project schema using $\prec_{\mathcal{R}}$ defined by the $\mathcal{G}^{\square}$ in Fig. 1. We abbreviate the method names for brevity. When p_2 receives works-on$(Alice, q_1)$, since works-on $\prec_{\mathcal{R}}$ delete-project, p_2 inserts works-on$(Alice, q_1)$ before delete-project(q_1), thereby preserving integrity. We return to this figure in § 5 when the notions of stable state σ_s, log splits ℓ_1 and ℓ_2, and split location c^* are defined.

Since local calls are always executed at the latest state of the replica, in order to respect the order of $\prec_{\mathcal{R}}$, once a call c is executed, only local calls that do not precede c in $\prec_{\mathcal{R}}$ can be accepted. In other words, when a call is executed, it *closes* the buckets of all methods that precede it in $\prec_{\mathcal{R}}$. For example, upon executing a call on delete-project, local calls on add-project and works-on are temporarily not accepted at that process. In Fig. 2, this happens when p_2 receives the local request add-project(r_2).

We need a way to *open* buckets again. To re-open them, *i.e.*, make their calls available again, calls later in the partial order must first be *stabilized.* A call is stabilized at a process when that process learns that there are no remote calls that will need to be inserted before it. (We elaborate on stability in § 5.) In Fig. 2, when the call on delete-project is stabilized, the buckets for add-project and works-on open again, resetting the sequence. We note that whether a bucket is open or closed is only relevant to local calls; remote calls are always executed at their designated bucket regardless.

However, receiving and executing remote concurrent calls in the $\prec_{\mathcal{R}}$ order is non-trivial. For example, consider the execution in Fig. 3. In this execution, p_3 receives the calls add-employee and works-on on from p_2. Then, p_3 receives delete-employee from p_1, which notably is concurrent to both calls from p_2. To respect (the new) $\prec_{\mathcal{R}}$, delete-employee should come after works-on, but also before add-employee, which is impossible. The solution is to first stabilize add-employee at p_2, so that p_2 receives delete-employee before executing works-on. Then, delete-employee will no longer be concurrent with works-on and p_3 does not need to execute delete-employee after works-on to respect $\prec_{\mathcal{R}}$. (We return to this example in § 4 and elaborate the solution.)

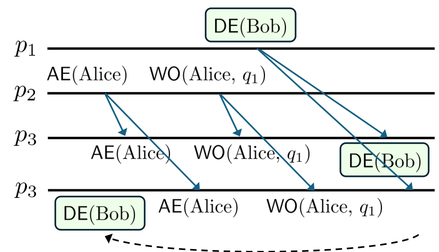

Fig. 3. The paradox of $\prec_{\mathcal{R}}$ order for concurrent calls. Let works-on $\prec_{\mathcal{R}}$ delete-employee $\prec_{\mathcal{R}}$ add-employee.

3 Replication Model

We now define objects, the interface and specification of replication protocols, and conflict relations.

Objects. An object is specified by a triple $O = \langle \sigma_0, \mathcal{I}, \mathcal{M} \rangle$, where σ_0 is the initial state of the object, $\mathcal{I}$ is the invariant (or integrity property), and $\mathcal{M}$ is the set of methods executable on the

object. The integrity predicate $\mathcal{I}$ is an invariant on the state. We saw an example invariant in the form of referential integrity in the schema above: every project identifier in WORKS must refer to a project in PROJECTS. The initial state satisfies the invariant $\mathcal{I}$. A method $m \in \mathcal{M}$ is a function from a parameter and pre-state, to a post-state and a return value. A call $c = m(v)$ on method m is the application of m to argument v. For two calls c_1 and c_2, if $c_1(\sigma) = \langle \sigma', v \rangle$, then the composition $c_1 \circ c_2$ is defined as $c_2(\sigma')$. Composition is naturally lifted to a sequence of calls. A complete call $\langle c : v \rangle$ is a pair of a call c and a return value v. The sequential specification of an object O is the set of all sequences of complete calls that are generated by the composition of any sequence of calls on O. More precisely, the sequential specification of an object O is the set of sequences of complete calls $\langle c_i : v_i \rangle$ such that for all $i \geq 0$, the pre-state of c_i is σ_i and $c_i(\sigma_i) = \langle \sigma_{i+1}, v_i \rangle$.

Replicated System. A protocol replicates the object across a set of processes P. Each process stores a concrete state Σ that represents a state σ of the object. When clear from the context, we simply call the object state σ that the concrete state Σ of a process represents, the state of that process. We consider a fail-stop model. Processes can arbitrarily crash without recovery. A correct process does not crash. We use a failure detector that guarantees that a process is suspected by correct processes if and only if it has crashed.

Protocol Interface. A client can send a request $\mathsf{call}(c)$ at any process $p \in P$ to execute the call c. We use *home*(c) to denote the process where call c is requested. Calls are propagated from the home to other processes. We refer to calls whose home is the current process as *local* calls, and to calls whose home is a different process as *remote* calls.

Upon receiving a request for a call c, a process responds with either $\mathsf{tentative\text{-}return}(c : v)$ or $\mathsf{not\text{-}accept}(c)$. If the process responds with the former, it also (later) issues a $\mathsf{commit}(c)$ response for the same call. If the process cannot accept c, it issues the $\mathsf{not\text{-}accept}(c)$ response. (For example, if the call violates the integrity property, it cannot be accepted.) Otherwise, the process accepts c and executes it. A process may re-execute c until it eventually commits c. Thus, the protocol issues the responses $\mathsf{tentative\text{-}return}(c : v)$ and $\mathsf{commit}(c)$ to return the value v when it tentatively executes, and finally commits c respectively. We say that a process has *executed* a call c if it has issued $\mathsf{tentative\text{-}return}(c : v)$. We say that a process has committed a call c if it has issued $\mathsf{commit}(c)$. We note that a return response induces a complete call $\langle c : v \rangle$. The *history* of an execution is the sequence of all requests and responses to and from all processes. The history of a process is the sequence of requests and responses to and from that process.

Protocol Specifications. A replication protocol must satisfy certain safety and liveness specifications: namely, well-formedness, object compliance, convergence and integrity for safety, and acceptance and propagation for liveness, which we define in turn.

Response Well-formedness. The following three properties enforce the semantics of responses.

DEFINITION 1 (RESPONSE-MATCHED). *A protocol is response-matched if no process issues a response for a call unless a process previously received a request for that call. Further, no process issues both a* not-accept*, and a* tentative-return *response for a call. Moreover, no process issues a* commit *before a* tentative-return *response for a call.*

DEFINITION 2 (COMMIT-FINAL). *A protocol is commit-final if no process issues* tentative-return *for a call it has previously issued* commit *for.*

DEFINITION 3 (COMMIT-UNIQUE). *A protocol is commit-unique if no process issues two* commit *responses for a call.*

Object Compliance. The protocol replicates a given object O. When a process executes a call c and returns a response, it is expected that the process updates its local state to represent the state change by c on O. As a process executes a sequence of calls, the return values of the responses

should match the return values of the corresponding sequence within the sequential specification of O. Even if processes re-execute calls, this correspondence should hold with the most recent response for each call. We formalize this property as the notion of *compliance*. We first establish a few definitions. The *recent history* h_r of a history h is the sub-history of h composed of the last tentative-return response of every call that appears in h. For example, consider the history $h = \mathsf{call}(c_1), \mathsf{tentative\text{-}return}(c_1 : v_1^1), \mathsf{call}(c_2), \mathsf{tentative\text{-}return}(c_1 : v_1^2), \mathsf{tentative\text{-}return}(c_2 : v_2)$, $\mathsf{call}(c_3), \mathsf{not\text{-}accept}(c_3)$. Then, $h_r = \mathsf{tentative\text{-}return}(c_1 : v_1^2), \mathsf{tentative\text{-}return}(c_2 : v_2)$. Notice that h_r induces a sequence of complete calls: $\langle c_1 : v_1^2 \rangle$, $\langle c_2 : v_2 \rangle$. A history *complies* with an object O if the sequence of complete calls induced by its recent history is in the sequential specification of O.

DEFINITION 4 (OBJECT COMPLIANCE). *A protocol complies with an object O if the history of every process complies with O.*

Convergence. Out-of-order execution of calls at different processes can lead to divergence in their states. A replication protocol should have processes converge to the same state.

DEFINITION 5 (CONVERGENCE). *A protocol is convergent if every pair of processes that execute the same set of calls have the same final state.*

Integrity. In addition to convergence, the protocol must maintain the integrity $\mathcal{I}$ of the object. The implementation of the body of each method assumes and relies on the invariant $\mathcal{I}$ in the pre-state. The post-state of a call is the pre-state of the next call. Therefore, a call should preserve the invariant in its post-state. A call c has integrity if its post-state satisfies the invariant $\mathcal{I}$, in which we say c is *permissible*. A sequence of calls has integrity if starting from the initial state, any prefix of the sequence results in a post-state that satisfies $\mathcal{I}$. An execution history of a process has integrity if the sequence of calls induced by its recent history has integrity.

DEFINITION 6 (INTEGRITY). *A protocol has integrity if the execution history of every process has integrity.*

Liveness. In addition to the above safety properties, there are liveness properties. The first is to respond to every requested call.

DEFINITION 7 (TERMINATION). *A protocol is terminating if every correct process eventually issues a response to every request.*

The second is to accept requested calls into the system when possible. If the requested call is not permissible in the local state of the process, the process is expected not to accept the call as it would violate integrity. For example, the call $\mathsf{WorksOn}(Alice, x)$ cannot be accepted if $Alice \notin \textsc{Employees}$. However, a process should eventually accept and execute a requested call if the call remains permissible.

DEFINITION 8 (ACCEPTANCE). *A protocol is accepting if every call that is requested infinitely often at a correct process where the call is permissible, then that process eventually accepts the call.*

This property can also be understood as the non-triviality condition as the other properties are trivially satisfiable by not executing any calls. The third liveness property guarantees that if a correct process executes a call, then the call is eventually replicated throughout the system, *i.e.*, all correct processes eventually execute that call.

DEFINITION 9 (PROPAGATION). *A protocol is propagating if every call accepted by a correct process is eventually executed by every correct process.*

We say a replication protocol is *proper* if every execution history of the protocol satisfies the above safety and liveness properties. This paper is interested in two additional properties: non-abortion and optimism. It adversely affects the client if a protocol executes a call and responds with a return value, but later aborts the call. In contrast to previous works, the non-abortion property requires that once a call is executed, it is eventually committed.

Definition 10 (Non-aborting). *A protocol is non-aborting if every correct process eventually commits every call that it executes.*

We note that in the non-aborting property, the condition for a call is execution not acceptance. Only the home process of a call accepts it, but any process (including remote processes) can execute it (*i.e.*, issue a tentative-return event for it).

In pessimistic protocols, upon receiving a call request, a process synchronizes with other processes before issuing a response. In contrast, optimistic protocols require a process to decide whether to accept a call based on its local state, without communicating with other processes.

Definition 11 (Optimism). *A protocol is optimistic if in every execution when only a single process takes steps, then that process eventually issues a response to every request it receives.*

Commutativity and Conflicts. With convergence and permissibility defined, two types of conflicts can occur between two calls: state-conflicts and permissibility-conflicts. We capture both conflict types as the negation of their commutativity counterparts. (Some basic definitions of commutativity are adopted from CRDTs [60] and well-coordination [32].)

State Commutativity and Conflict. Two calls state-commute if swapping their execution order does not affect the resulting post-state. That is, c_1 and c_2 state-commute if $c_1 \circ c_2(\sigma) = c_2 \circ c_1(\sigma)$, for *every* pre-state σ. Otherwise, the two calls *state-conflict.* This can be lifted to methods. Two methods m_1 and m_2 state-conflict if there are *any* two calls on m_1 and m_2 that state-conflict. We represent state-conflicting methods as an undirected graph, possibly with loops, $\mathcal{G}_S$: the vertex set is $\mathcal{M}$ and (m, m') is an edge if m and m' state-conflict. We call $\mathcal{G}_S$ the *state-conflict graph* (or state graph for short). The next theorem [32] about state-conflicts and convergence will be useful later.

Theorem 1. *If two sequences of a set of calls have the same order for every pair of state-conflicting calls then their execution from the same pre-state results in the same post-state.*

Permissible Commutativity and Conflict. The state-commutativity relation allows calls to change in their order without altering their post-state, thereby preserving convergence. We define a similar relation for permissibility, allowing calls to change in their order without violating their permissibility, thereby preserving integrity.

A call permissible-right-commutes ($\mathcal{P}_R$-commutes) with another call if the former call retains its permissibility if moved after the latter call. Formally, c_2 $\mathcal{P}_R$-commutes with c_1 if for every state σ, if c_1 and c_2 are both permissible in σ, then c_2 is permissible in $c_1(\sigma)$. Otherwise, we say c_2 permissible-right-conflicts ($\mathcal{P}_R$-conflicts) with c_1. Note that, unlike state-conflicts, the $\mathcal{P}_R$-conflict relation is not symmetric. Similar to state-commutativity, we can lift this to methods. Further, we can represent $\mathcal{P}_R$-conflicting methods as a directed graph, possibly with loops, $\mathcal{G}_\mathcal{P}$: the vertex set is $\mathcal{M}$, and (m, m') is an edge if m $\mathcal{P}_R$-conflicts with m'. We call $\mathcal{G}_\mathcal{P}$ the *permissibility-conflict graph* (or permissibility graph for short).

4 ORDTs

In this section, we define ORDTs (Optimistically Replicated Data Types). Then, we define the static partial order of their methods, its properties, and make observations that inform the design of the upcoming protocol.

We remember that given the $\mathcal{G}_S$ and $\mathcal{G}_{\mathcal{P}}$ of an object O, a conflict graph $\mathcal{G}^{\square}$ of O is a graph with the following properties. The vertex set of $\mathcal{G}^{\square}$ is $\mathcal{M}$, the same as $\mathcal{G}_S$ and $\mathcal{G}_{\mathcal{P}}$. Every (directed) edge in $\mathcal{G}_{\mathcal{P}}$ is in $\mathcal{G}^{\square}$. At least one directed variant of every (undirected) edge in $\mathcal{G}_S$ is in $\mathcal{G}^{\square}$.

Definition 12 (ORDT). *An Optimistically Replicated Data Type (ORDT) is an object with an acyclic and loop-free conflict graph.*

For brevity, we say acyclic instead of both acyclic and loop-free. Going forward, we assume we have an acyclic $\mathcal{G}^{\square}$, from which we can define the desired partial order $\prec_{\mathcal{R}}$. Define $\prec_{\mathcal{R}}$ to be a partial order over $\mathcal{M}$ where $m \prec_{\mathcal{R}} m'$ if m can reach m' in $\mathcal{G}^{\square}$, excluding trivial self-reachability. Note then, $\prec_{\mathcal{R}}$ is irreflexive, asymmetric, and transitive, since $\mathcal{G}^{\square}$ is acyclic. Thus, $\prec_{\mathcal{R}}$ is a strict partial order over methods. We write $m \succ\!\prec_{\mathcal{R}} m'$ if two methods are comparable under $\prec_{\mathcal{R}}$, *i.e.*, either $m \prec_{\mathcal{R}} m'$ or $m' \prec_{\mathcal{R}} m$ holds, and we wite $m \prec\!\succ_{\mathcal{R}} m'$ if they are incomparable. Additionally, we say m is *prior-or-incomparable* with m' if m precedes *or* is incomparable to m', *i.e.*, if either $m \prec_{\mathcal{R}} m'$ or $m \prec\!\succ_{\mathcal{R}} m'$, which we write as $m \prec\!\prec\!\succ_{\mathcal{R}} m'$. The order can be simply lifted from methods to calls: for any pair of calls c and c' on methods m and m', $c \prec_{\mathcal{R}} c'$ if $m \prec_{\mathcal{R}} m'$.

We will now make informal observations about the replicated execution of ORDTs which will inform the design and formal reasoning for the protocol Frashokereti.

Lemma 1. *For any state σ and pair of calls c_1 and c_2, if c_1 and c_2 are permissible in σ, and $c_1 \prec\!\prec\!\succ_{\mathcal{R}} c_2$, then c_2 is permissible in $c_1(\sigma)$.*

Proof. Since $c_1 \prec\!\prec\!\succ_{\mathcal{R}} c_2$, by definition, c_2 $\mathcal{P}_R$-commutes with c_1. Using the assumption that c_2 is permissible in σ, by commuting c_2 after c_1, we get that c_2 is permissible in $c_1(\sigma)$. □

Intuitively, this means that so long as $c_1 \prec\!\prec\!\succ_{\mathcal{R}} c_2$, we can insert c_1 before c_2, and c_2 will remain permissible. We can generalize c_1 to any sequence s of calls: if c_2 is permissible in σ, and for every call c_1 in s, $c_1 \prec\!\prec\!\succ_{\mathcal{R}} c_2$, then c_2 is permissible in $s(\sigma)$.

For the second observation, say a sequence of calls s *respects* $\prec_{\mathcal{R}}$ if for every pair of calls c_1, c_2 in s, if c_1 precedes c_2 in s, then c_1 is prior-or-incomparable with c_2.

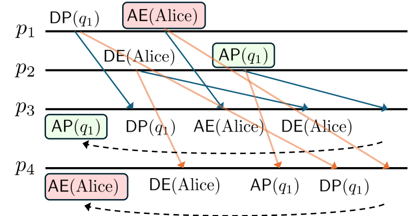

Fig. 4. Observation 1: Ordering concurrent calls by $\prec_{\mathcal{R}}$. Let add-employee $\prec_{\mathcal{R}}$ delete-employee and add-project $\prec_{\mathcal{R}}$ delete-project.

Lemma 2. *For every set of calls C where every call is permissible in a pre-state σ, all permutations of C that respect $\prec_{\mathcal{R}}$ executed upon σ converge to the same post-state and preserve permissibility for each call.*

Proof. (Convergence) Let π_1 and π_2 be two permutations of C that respect $\prec_{\mathcal{R}}$. By Theorem 1, it is sufficient to show every pair of state-conflicting calls appear in the same order in both π_1 and π_2. If two calls c and c' in C state-conflict, then either $c \prec_{\mathcal{R}} c'$ or $c' \prec_{\mathcal{R}} c$ holds. Since π_1 and π_2 respect $\prec_{\mathcal{R}}$, the calls c and c' must appear in the same order across π_1 and π_2.

(Permissibility) Let π be a permutation of C that respects $\prec_{\mathcal{R}}$. Consider some call c_2 in π and the sequence s of calls that precedes it. Since c_2 is permissible in σ, and for every call c_1 in s , $c_1 \prec\!\prec\!\succ_{\mathcal{R}} c_2$, by the generalization of Lemma 1 above, c_2 is permissible in $s(\sigma)$. □

Intuitively, this means that if processes that share a common pre-state execute a set of concurrent calls, they can order the calls according to $\prec_{\mathcal{R}}$ to preserve both convergence and permissibility.

Thus, the protocol Frashokereti will order concurrent calls by the $\prec_{\mathcal{R}}$ order. For example, let add-employee $\prec_{\mathcal{R}}$ delete-employee, and add-project $\prec_{\mathcal{R}}$ delete-project, and consider Fig. 4. Process p_1 executes delete-project and then add-employee, and process p_2 executes delete-employee

and then add-project. Then, these calls are sent to process p_3. First, the two calls delete-project and add-employee from p_1, then the call delete-employee from p_2 are received and executed. We note that delete-employee is concurrent with the two previous calls, and within the order $\prec_{\mathcal{R}}$ we have add-employee $\prec_{\mathcal{R}}$ delete-employee which is respected in p_3. Finally, the call add-project is received which is also concurrent to the two first two calls, and within the order $\prec_{\mathcal{R}}$, we have add-project $\prec_{\mathcal{R}}$ delete-project. Therefore, in order to respect $\prec_{\mathcal{R}}$ in p_3, we find the first call in p_3 which is concurrent to add-project, and succeeds it in $\prec_{\mathcal{R}}$ to be delete-project, and insert add-project before it. Now consider the process p_4. The calls from p_2 and then the calls from p_1 are received by p_4. Similar to p_3, in order to respect $\prec_{\mathcal{R}}$ in p_4, the final call add-employee is inserted before delete-employee. We note that although p_3 and p_4 execute calls in different orders, since they preserve the order $\prec_{\mathcal{R}}$, they converge to the same state. We also note that although calls are delivered in the causal order, they are not necessarily executed in that order.

Observation 1. *In order to locally execute a remote call c, we should insert c before the first call c^* which is concurrent to c, and succeeds c in $\prec_{\mathcal{R}}$. (If there is no such c^*, append c at the end.)*

However, the fact that concurrency is not a transitive relation brings forth challenges for the strategy above. For example, consider again the execution in Fig. 3. In this execution, the calls add-employee and works-on from p_2, and then the call delete-employee from p_1 are received by p_3. The last call delete-employee is concurrent to both add-employee and works-on, but those two are not concurrent to each other. In order to respect $\prec_{\mathcal{R}}$, delete-employee should execute before add-employee, but also after works-on, which is impossible. If the two calls add-employee and works-on are reordered, delete-employee can be inserted in the middle, and $\prec_{\mathcal{R}}$ is respected. However, the permissibility of works-on is dependent on add-employee, and the reordering makes it impermissible. As Fig. 5 shows, the solution is to make p_2 receive the concurrent call delete-employee before executing works-on so that the two are not concurrent anymore. When the call works-on is requested at p_2 after add-employee, which succeeds works-on in $\prec_{\mathcal{R}}$, we should first make add-employee stable. The call add-employee is stable at p_2 when every process receives add-employee, and p_2 receives the concurrent call delete-employee. (We will elaborate stability and how it is implemented in the next section.) Thus, now when works-on is delivered to p_3, delete-employee is only concurrent to add-employee, and can be inserted before it. Thus, the order $\prec_{\mathcal{R}}$ is respected for concurrent calls.

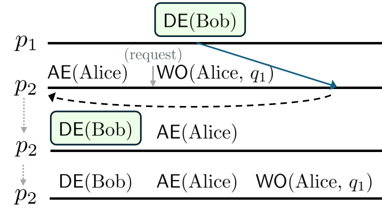

Fig. 5. Observation 2: Already executed calls that precede the requested call in $\prec_{\mathcal{R}}$ should be stable.

Observation 2. *In order to execute a local call c, the calls that are already executed and succeed c in $\prec_{\mathcal{R}}$ should be stable.*

5 Optimistic Protocol Frashokereti

In this section, we present our optimistic replication protocol, Frashokereti. It optimistically replicates objects with an acyclic conflict graph $\mathcal{G}^{\square}$. We split the presentation of our protocol into two parts. In the first part, we assume every process has access to a local oracle O, which we call the stabilizer. We will construct our protocol using this oracle. In the second part, we explain how to implement the stabilizer. As the name suggests, processes will use O to stabilize calls, *i.e.*, commit them. When a process invokes O on a call c, the stabilizer responds Yes if every process has executed c and the current process **self** has received all calls concurrent with c. Otherwise, the stabilizer responds No.

Logs. Every process p maintains a *log* ℓ_p of the calls it has executed to induce its current state σ_n. We write $c \in \ell$ to denote that call c is in log ℓ. A log induces a total precede order $\prec_\ell$ on its calls: $c \prec_\ell c'$ if c precedes c' in ℓ. A log ℓ also induces a sequence of states by executing the call sequence on the initial state σ_0. Additionally, a log ℓ is split into a prefix of *committed* calls and a postfix of *tentative* calls. Recall that tentative calls are subject to rollbacks (as a process can issue multiple tentative-return responses for the same call), whereas committed calls are not (a process cannot issue another response for a call that it has issued commit for).

Causal Ordering. The protocol uses causal broadcast to communicate between processes. We use the classical definition of the *causal* or happens-before relation [37]. For two calls c_1 and c_2, c_1 is causally before c_1, written $c_1 \rightarrow_{co} c_2$, if (1) the home of c_1 and c_2 is the same process p, and c_1 is executed before c_2 at p, (2) c_1 is executed at $home(c_2)$ before c_2 is executed at $home(c_2)$, or (3) there is a call c_3 where c_1 is causally before c_3, and c_3 is causally before c_2. We say c_1 and c_2 are *concurrent* if they are not related by the causal order, written $c_1 \parallel_{co} c_2$.

The causal relation can be captured using standard vector clocks [26]. Each process $p_i \in P$ maintains a local vector of integers of size $|P|$, initialized to the zero vector. Upon executing a local call, process p_i increments index i of its local vector clock. It then attaches this clock to the call before propagating it. Upon executing a remote call whose home process is p_j, the receiving process increments index j of its local vector clock. A vector clock v is less than another v' ($v < v'$), if $\forall i.\, v_i \leq v'_i \land \exists j.\, v_j < v'_j$. For two calls c and c' with vector clocks v and v', c is causally before c' iff $v < v'$. Thus, if neither $v < v'$ nor $v' < v$, then c and c' are concurrent. For the remainder of this paper, we assume messages are broadcast using reliable causal broadcast: if $c \rightarrow_{co} c'$ (based on the vector clocks of c and c' at the time of local execution), then every process receives c before c'.

Replication Protocol. FRASHOKERETI is presented in Alg. 1. Every process is initialized with the statically computed partial order $\prec_{\mathcal{R}}$. Rather than storing the entire log, each process only stores three salient variables: (1) the stable (committed) state σ_s, (2) the tentative postfix of the log ℓ, written $\overset{\circ\!-}{\ell}$, and (3) the current state σ_n. The stable state σ_s is the post-state of executing the committed prefix of the log on the initial state σ_0. The sequence $\overset{\circ\!-}{\ell}$ is the postfix of tentative calls in the log (hence the symbol). When a tentative calls in $\overset{\circ\!-}{\ell}$ becomes stable, its effects are applied to σ_s and removed from $\overset{\circ\!-}{\ell}$. Lastly, the current state σ_n is the post-state of executing the tentative log $\overset{\circ\!-}{\ell}$ on the stable state σ_s, *i.e.*, the post-state of the entire log ℓ. The current state is the most up-to-date state, although it contains tentative calls. Both σ_s and σ_n are initialized to σ_0, and $\overset{\circ\!-}{\ell}$ is initialized to the empty sequence.

A client can issue a request to execute a call at any process as its home process. FRASHOKERETI issues a response back to the client when the call is (re-)executed. The protocol consists of four handlers: two handlers for the local and remote calls, and two handlers for stabilizing calls.

LOCAL HANDLER. Upon receiving a local call c from a client, if the following two conditions hold then execute c on the current state σ_n to update it, append c to the tentative log $\overset{\circ\!-}{\ell}$, and issue a tentative-return response for c.

(LH.1) c is permissible at the stable state σ_s, and
(LH.2) for every call c' in $\overset{\circ\!-}{\ell}$, $c' \not\prec\!\!\succ_{\mathcal{R}} c$.

Otherwise, the protocol issues a not-accept response for the call. The client can explicitly request the call again if needed. We note that the client discretion is needed since some calls never become permissible. We note that the local handler aligns with Lemma 1 and Observation 2.

REMOTE HANDLER. Upon receiving a remote call c from another process,

(RH.1) Find the index i of the first call $c^* \in \overset{\circ\!-}{\ell}$ such that $c \parallel_{co} c^*$ and $c \prec_{\mathcal{R}} c^*$, if there is such a call. Otherwise, let i be $|\overset{\circ\!-}{\ell}|$.

Algorithm 1: Protocol FRASHOKERETI

1 **Implements:** Optimistic Replication
2 **request** : *Call*(c), **response** : not-accept(c), tentative-return$(c : v)$, commit(c)
3 **Vars:** $\prec_{\mathcal{R}}$ ▷ Partial order
4 $\sigma_s \leftarrow \sigma_0$ ▷ Stable state
5 $\overset{\circ\!-}{\ell} \leftarrow \varnothing$ ▷ Sequence of tentative calls
6 $\sigma_n \leftarrow \sigma_0$ ▷ Current (latest) state
7 **Uses:** *rcb* : ReliableCausalBroadcast
8 **request** : *broadcast*(c), **response** : *deliver*(c)
9 O : Stabilizer
10 **request** : *is-stable*(c), **response** : YES, NO
11 **upon request** *Call*(c) ▷ Local Handler
12 **if** *c permissible in σ_s and* ▷ (LH.1)
13 $\forall c' \in \overset{\circ\!-}{\ell}.\ c' \prec\!\!\!\succ_{\mathcal{R}} c$ ▷ (LH.2)
14 **then**
15 $\langle \sigma_n, v \rangle \leftarrow c(\sigma_n)$
16 append c to $\overset{\circ\!-}{\ell}$
17 *rcb* **request** *broadcast*(c)
18 **response** tentative-return$(c : v)$
19 **else**
20 **response** not-accept(c)
21 **upon** *rcb* **response** *deliver*(c) ▷ Remote Handler
22 $i \leftarrow$ index of first $c^* \in \overset{\circ\!-}{\ell}$ such that
23 $c \parallel_{co} c^* \wedge c \prec_{\mathcal{R}} c^*$ ▷ (RH.1)
24 otherwise, $|\overset{\circ\!-}{\ell}|$
25 $\ell_1 \leftarrow \overset{\circ\!-}{\ell}[0, ..., i)$ ▷ (RH.2)
26 $\ell_2 \leftarrow \overset{\circ\!-}{\ell}[i, ...]$
27 $\overset{\circ\!-}{\ell} \leftarrow$ concatenate(ℓ_1, c, ℓ_2)
28 $\sigma_n \leftarrow \sigma_s$ ▷ (RH.3)
29 **foreach** c' *in* ℓ_1 **do**
30 $\langle \sigma_n, _ \rangle \leftarrow c'(\sigma_n)$
31 $\langle \sigma_n, _ \rangle \leftarrow c(\sigma_n)$
32 **foreach** c' *in* ℓ_2 **do**
33 $\langle \sigma_n, v \rangle \leftarrow c'(\sigma_n)$
34 **response** tentative-return$(c : v)$
35 **upon periodically**
36 O **request** *is-stable*$(\overset{\circ\!-}{\ell}[0])$
37 **upon** O **response** YES
38 **response** commit$(\overset{\circ\!-}{\ell}[0])$
39 $\sigma_s \leftarrow \overset{\circ\!-}{\ell}[0](\sigma_s)$
40 $\overset{\circ\!-}{\ell} \leftarrow \overset{\circ\!-}{\ell}[1...]$

(RH.2) Split $\overset{\circ\!-}{\ell}$ at i into a prefix ℓ_1 and suffix ℓ_2 such that ℓ_1 is the subsequence preceding i, and ℓ_2 is the subsequence containing and following i.
(RH.3) (Re-)execute calls in order of ℓ_1, the call c, and then ℓ_2 on σ_s to calculate the new state σ_n.[2]

When re-executing the calls of ℓ_2 in (RH.3), issue a new tentative-return response for that call.

Intuitively, the call c is placed amongst the calls in $\overset{\circ\!-}{\ell}$ that are concurrent to c. The exact position of c between these calls is decided by the order $\prec_{\mathcal{R}}$. We note that the remote handler aligns with Observation 1.

STABILIZER. Periodically invoke O on the first call c in the tentative log $\overset{\circ\!-}{\ell}$. As previously mentioned, O is a *local oracle* and processes invoke it asynchronously. If O returns YES, apply c to the stable state, and remove it from $\overset{\circ\!-}{\ell}$. After doing so, issue a commit response for c.

In Fig. 2, we saw an example of the protocol for the employee-project schema where process p_2 inserts the remote call works-on$(Alice, q_1)$ before delete-project(q_1).

Stabilization. The replication protocol uses a local oracle, the stabilizer O, to commit calls. We present an implementation of O. The stabilizer tracks *causal stability* [12, 37]. A process p learns that a call c is causally stable upon receiving from every process a call whose vector clock is larger than the vector clock of c. The stabilizer O responds YES for the stability of the call c when this condition holds, and NO otherwise. By causal delivery, this condition implies that every process must have (received and) executed c as well. Additionally, it implies that there are no calls concurrent to c that are in-transit to p. Consider some call c' that is concurrent to c. Thus, at $home(c')$, c' is executed before c. The process p has received a call that is causally after c from $home(c')$. Thus, p also received all calls executed before c at $home(c')$. Therefore, p must have

[2] In fact, it is enough to re-execute calls after ℓ_1. However, we do not assume the post-state of ℓ_1 is stored. In settings where states are small and enough memory is available, re-execution can be avoided by storing this state.

received c'. We note that the causal stability of a call is a process-local property, meaning that at any given time, a call may be causally stable at some process, but not another.

Importantly, a process must hear from every process to stabilize a call, bringing forth two points. First, if a process is never the home process of any call, it never shares its local vector clock with other processes. This is resolved by having processes periodically execute a *proxy call* to broadcast their local vector clock, if they have not executed a local request for some time. Proxy calls are not real calls, *i.e.*, they do not change the state, and are always executable under the local handler conditions, but the corresponding vector clock index is incremented when delivered. Second, a process stabilizes a call after hearing from every correct process (rather than every process). Thus, processes use the failure detector to determine which processes it must hear from to stabilize a call.

6 Correctness of FRASHOKERETI

In this section, we prove the correctness of FRASHOKERETI as described in the previous section. The intuition and full proofs for each specification property are available in the Appendix ?? and ??. At the end of this section, we consider the message and local time complexity of FRASHOKERETI.

THEOREM 2. *Every ORDT has a proper, non-aborting, optimistic protocol.*

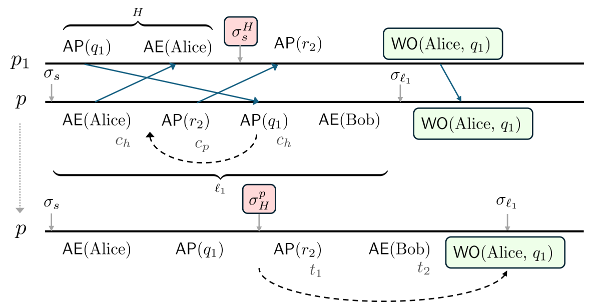

Fig. 6. An example execution of the employee-project schema illustrating how permissibility of a remote call is derived from permissibility in its home process, as explained in the proof of integrity, ??. We use $\prec_{\mathcal{R}}$ defined by the $\mathcal{G}^{\square}$ in Fig. 1.

We illustrate one of our proofs here, specifically the proof of the integrity property, with an example in Fig. 6. Whenever a process executes a call, it adds that call to its log. Thus, to show FRASHOKERETI preserves integrity, we consider any log that it can generate. The argument is inductive on an incoming call c: after inserting c into the local log, we argue that every call is permissible in its (new) pre-state. For this sketch, we consider only when c is a remote call (*i.e.*, works-on($Alice, q_1$) from p_1). The remote handler splits the tentative log of the current process p into two parts, ℓ_1 and ℓ_2, and c is inserted between. We show (1) c is permissible in the post-state of ℓ_1 (called σ_{ℓ_1}), and (2) every call in ℓ_2 is permissible in its new pre-state.

The proof of (1) is done in two steps. First, we derive permissibility for c from its local execution at $home(c)$, that is, p_1. We reconstruct the stable state σ_s^H of p_1 (at the time of executing c) at the current process p, without altering σ_{ℓ_1}, the pre-state of c. To argue this, let H be the set of calls stable for p_1 (*i.e.*, add-project(q_1) and add-employee(Alice)). We show for every call c_h in H, and every call c_p in p but not in H (*i.e.*, add-project(r_2)), if $c_p \prec_{\ell_p} c_h$, then c_h and c_p state-commute. This allows us to state-commute every call c_h before every c_p call without changing σ_{ℓ_1}. We can re-arrange the calls so that the first $|H|$ calls of p are the calls of H resulting in a state σ_H^p. By convergence, which we prove later, σ_H^p is equal to σ_s^H. By (LH.1), c must be permissible in this state σ_s^H and thus, in σ_H^p.

The second step is to show c $\mathcal{P}_R$-commutes with every call between this state and σ_{ℓ_1}. These calls are one of two types: (t_1) calls causally before c (*i.e.*, add-project(r_2)) or (t_2) calls concurrent to c (*i.e.*, add-employee(Bob)). Calls of type t_1 are in the tentative log of p_1, and by (LH.2), c must $\mathcal{P}_R$-commute with them. By (RH.1), every t_2 call preceding σ_{ℓ_1} must be prior-or-incomparable with

c. Thus, c is permissible in σ_{ℓ_1}. To show (2), we repeat a similar argument to the one above for each call in ℓ_2. The part ℓ_2 is empty in this example.

Complexity. To complete this section, we consider both the message complexity and local time complexity of Frashokereti. The latter is especially relevant as existing protocols have exponential-time local complexity. Additionally, we show later in § 8 that proper, optimistic protocols for general replicated objects must (repeatedly) solve an NP-Complete problem.

Message Complexity. Our protocol relies on reliable causal broadcast to propagate messages between processes. In the worst case, this yields $O(n^2)$ messages per call, where n is the number of processes. For stability, if calls are issued across processes uniformly (and often), then proxy calls are never issued. In comparison, in the worst case, all requests are made at a single process, resulting in an additional $O(n^2)$ messages for each of $n - 1$ proxy calls. The entire execution exchanges $O(k \cdot n^3)$ messages where k is the total number of calls. Reliable broadcast takes $O(f)$ rounds when f processes crash and 2 rounds otherwise. The execution takes $O(k + n)$ rounds.

Local Complexity. Inspection of our protocol handlers shows Frashokereti has linear time complexity with respect to the length of the tentative log, per call. Assume executing a given call takes $O(1)$ time. Let l be the maximum length of any tentative log; in the worst case, this is k. The local handler runs in $O(l)$ time, as checking permissibility of the local request in (LH.1) takes constant time, and checking prior-or-incomparability across the entire tentative log in (LH.2) takes $O(l)$ time. The remote handler also runs in $O(l)$ time: finding c^* in (RH.1) and re-executing the tentative log in (RH.2) both take $O(l)$ time. Therefore, Frashokereti runs in $O(l)$ local time per call and $O(l \cdot k)$ time over the entire execution.

7 The ORDT Class of Objects

In this section, we show that ORDTs subsume CRDTs. We further present method abstraction and transformation techniques to extend the applicability of ORDTs. We apply them to show that (abstracted) and transformed relational schema fall under the ORDT class. As relational schemata are the prevalent approach for data representation and manipulation, this shows the general applicability of ORDTs.

7.1 Subsuming CRDTs

In this subsection, we show the ORDT class indeed generalizes Conflict-free Replicated Data Types (CRDTs). Additionally, we show that if an object qualifies as a CRDT, the execution of Frashokereti is indistinguishable from its CRDT execution, in its properties and order of calls. Recall that CRDTs characterize the class of replicated objects with neither state nor permissibility conflicts. Formally, an object can be cast as a CRDT if (c.1) all methods state-commute[3] and (c.2) it has no invariant.

Theorem 3. *Every CRDT is an ORDT.*

An object is a CRDT only if it has an empty (edgeless) conflict graph $\mathcal{G}^{\square}$. This is because by (c.1), the state graph $\mathcal{G}_S$ of O is empty. Further, by (c.2), every call is permissible in every state, and thus $\mathcal{G}_{\mathcal{P}}$ is also empty. Therefore, $\mathcal{G}^{\square}$ is empty and thus acyclic, meaning O is also an ORDT.

Since every CRDT is also an ORDT, every CRDT admits Frashokereti. We compare the execution of the object as a CRDT versus Frashokereti, and show they are indistinguishable in their guarantees and order of calls. CRDTs guarantee the properties of strong eventual consistency: convergence and propagation. We prove Frashokereti guarantees both of these properties in Lemmas **??** and **??**. We turn our attention to the order of calls in a CRDT: (1) a CRDT can execute any

[3]In this paper, we consider operation-based CRDTs. In its original presentation, CRDTs require calls that may be issued *concurrent* to (state-)commute. However, in this work, we are interested in general clients.

local request and (2) all calls are executed at the current (latest) state. We show that both of these hold under FRASHOKERETI as well. Semantically, this means if an object is a CRDT, FRASHOKERETI never re-executes calls. Note that since $\mathcal{G}^{\square}$ is empty, all methods are incomparable under $\prec_{\mathcal{R}}$.

First, every process executes every local request. Consider some local call c, and the conditions of the local handler. By (c.2), the object has no invariants. Therefore, every call is permissible in every state, notably the stable state, *i.e.*, (LH.1) holds for c. Further, for every call c' in the tentative log, we have $c' \not\prec_{\mathcal{R}} c$ since no methods are related under $\prec_{\mathcal{R}}$, implying (LH.2) also holds.

Second, we show every call is executed at the current state. This trivially holds for local calls, so we only need to consider remote calls. Consider some remote call c under the execution of the remote handler. The handler executes c right before the c^* call found by (RH.1), if it exists; otherwise it executes c at the current state. Since all methods are incomparable under $\prec_{\mathcal{R}}$, there cannot be a c^* such that $c \prec_{\mathcal{R}} c^*$. Therefore, c is executed at the current state.

Remark: By showing all calls are executed at the current state (2), we actually show the stabilizer is unnecessary in the case the object is a CRDT. A call will never be inserted before another call, therefore no call is tentative and thus immediately stable upon execution.

7.2 Abstraction and Transformation

We remember that ORDTs are defined as objects with an acyclic and loop-free conflict graph $\mathcal{G}^{\square}$. In this subsection, we discuss two techniques that allow us to remove conflicts and make conflict graphs acyclic and loop-free. Intuitively, the first technique, that we call *abstraction*, can take an object O with a *cyclic* conflict graph and construct a new object O^A with an *acyclic* conflict graph, such that O^A can simulate the behavior of O. Then, O^A enjoys the benefits of being an ORDT, while behaving as O. The second technique *transforms* the implementation of methods to repair integrity in the post-state when the original methods are impermissible. Thus, it reduces permissible conflicts. We describe each technique in turn, with examples.

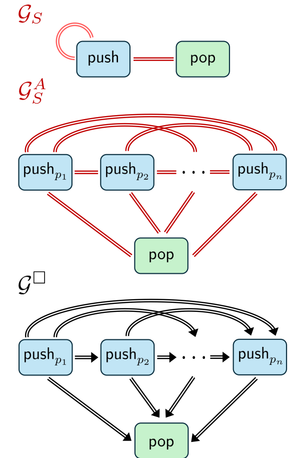

Fig. 7. Top: the state-conflict graph for a *stack* object. Middle: the loop on the push method is split into n vertices as described by our procedure. Bottom: one possible construction of $\mathcal{G}^{\square}$.

Abstraction. We define abstraction between sets of methods and between objects.

DEFINITION 13 (ABSTRACTION). *For two sets of methods $\mathcal{M}$ and $\mathcal{M}^A$ on some object, $\mathcal{M}^A$* abstracts *$\mathcal{M}$ if for every pre-state σ, for every call c on some method $m \in \mathcal{M}$, there exists a call c^A on some method $m^A \in \mathcal{M}^A$, such that $c(\sigma) = c^A(\sigma)$.*

This can be lifted to objects: object O^A abstracts object O if they have the same initial state σ_0, the same invariant $\mathcal{I}$, and the set of methods for O^A abstracts the set of methods for O. In this paper, to derive abstract method sets, we split a method into multiple methods without changing the method body, which can redirect edges and remove cycles. This splitting can remove superfluous conflicts. We showcase the idea in the proof of the following lemma.

LEMMA 3. *Let object $O = \langle \sigma_0, \mathcal{I}, \mathcal{M} \rangle$ have a state graph $\mathcal{G}_S$ that contains loops. There exists an object $O^A = \langle \sigma_0, \mathcal{I}, \mathcal{M}^A \rangle$ that abstracts O and has a loop-free state graph $\mathcal{G}_S^A$.*

PROOF. Consider some method $m \in \mathcal{M}$ with a loop in $\mathcal{G}_S$. The object that we construct is parametric with the number of processes. We split m into per-process methods: replace m with

n-many methods $m_{p_1}, \ldots, m_{p_n}$, where n is the number of processes in the system. Every m_{p_i} has the same method body as m. A call on m at the home process p_i is mapped to m_{p_i}. We recompute $\mathcal{G}_S^A$ with the split methods: edges that were not incident to m are the same; edges that were incident to m are now incident to each m_{p_i}; every pair of newly added vertices share an edge. Importantly though, no m_{p_i} has a loop. We verify that $\mathcal{M}^A$ abstracts $\mathcal{M}$. First, $\mathcal{M}^A = \mathcal{M} \setminus \{m\} \cup \{m_{p_1}, \ldots, m_{p_n}\}$. For any call c on a method in $\mathcal{M} \setminus \{m\}$, that method still exists in $\mathcal{M}^A$. For any call c on m at p_i, m_{p_i} yields the same post-state as m since m and m_{p_i} have the same method body. We note that the size of the graph does not necessarily translate to more or less behaviors. □

As an example, Fig. 7 shows $\mathcal{G}_S$ and its split version. Let us consider how this per-process abstraction affects the permissibility graph $\mathcal{G}_\mathcal{P}$. In the resulting permissibility graph $\mathcal{G}_\mathcal{P}^A$, since m_{p_i} has the same method body as m: any edge not incident to m are as before; edges that were incident to m are now incident to each m_{p_i}, in the same direction. We observe that if m does not have a loop in $\mathcal{G}_\mathcal{P}$, newly added vertices do not have edges to each other. (On the other hand, if m had a loop, every newly added vertex has an edge to and from every other newly added vertex.) Thus, if $\mathcal{G}_\mathcal{P}$ is acyclic and loop-free, $\mathcal{G}_\mathcal{P}^A$ is acyclic and loop-free as well. This observation together with Lemma 3 will help to prove the following lemma.

Lemma 4. *If an object O has an acyclic and loop-free permissibility graph, then there exists an ORDT that abstracts O.*

Proof. Let O be an object with an acyclic and loop-free $\mathcal{G}_\mathcal{P}$. By the per-process method splitting procedure of Lemma 3, we have an object O^A such that (1) O^A abstracts O, (2) the state graph $\mathcal{G}_S^A$ of O^A is loop-free, and by the observation above, (3) the permissibility graph $\mathcal{G}_\mathcal{P}^A$ is acyclic and loop-free. We construct an acyclic and loop-free conflict graph $\mathcal{G}^\square$ for O^A. First, set $\mathcal{G}^\square$ to $\mathcal{G}_\mathcal{P}^A$. By (3), compute a topological sort T over $\mathcal{G}_\mathcal{P}^A$. For every edge between m and m' in $\mathcal{G}_S^A$, let m precede m' in T; add directed edge (m, m') to $\mathcal{G}^\square$ (if it does not already exist). By (2) and the fact that edges are added along T, then $\mathcal{G}^\square$ is acyclic and loop-free. Thus, O^A is an ORDT. □

In addition to splitting methods by process identifiers, we can split methods by their *parameter domains*. The idea is to split a method into multiple methods according to the sub-domains of the input parameters. To showcase this, consider again the employee-project schema object we saw in § 2. Suppose we introduce a new method called replace(S, e_1, e_2), where S is a set in {Employees, Projects, Works}, e_1 is an element in S, and e_2 is the element that will replace e_1. The replace method shares the cascade behavior of delete: if replacing an employee or project (as primary key), all tuples in works-on adopt the change as well (as foreign key). We observe that replace $\mathcal{P}_R$-conflicts with itself by considering the two following calls on replace: $r_1 = (\text{works-on}, \langle Alice, q_1\rangle, \langle Alice, q_2\rangle)$ and $r_2 = (\text{Projects}, q_2, q_3)$. Semantically, r_1 is re-assigning *Alice* to work on project q_2 instead of q_1, while r_2 is renaming project q_2 to q_3. Replacement r_1 $\mathcal{P}_R$-conflicts with r_2: since q_2 no longer exists in the post-state of r_2, if r_1 is executed after r_2, referential integrity is violated.

Let us refer to the employee-project schema equipped with replace as O. We construct an object O' that has an acyclic and loop-free permissibility graph which abstracts O. Object O' is the same as O, except we split the replace method into three methods, depending on the set being modified. That is, we split replace into replace-employee, replace-project, and replace-works. Each new method is executed if the specified set S matches its name. If we check again, replace-works $\mathcal{P}_R$-conflicts with replace-employee and replace-project, but not vice-versa. The permissibility graph of the split object is acyclic and loop-free. Further, since the union of the domains for the new replace methods covers the entire domain of the original replace method, we see that O' indeed abstracts O. Thereby Lemma 4, there exists an ORDT that abstracts O', and further, abstracts O.

Transformations. The tenet of abstraction is that the semantic behavior of the methods do not change. However, abstraction is sometimes not sufficient to convert an object into an ORDT. We further present another technique, called *transformation*, in which the semantic behavior of the methods are transformed. In order to resolve conflicts, *i.e.*, remove edges from the conflict graphs, the idea is to incorporate additional behavior to "repair" the invariant when conflicts do occur.

We introduce two types of transformations that can generally be applied to any method: *yield* and *force*. If a call on the original method is permissible, then the yield/force method executes the original method normally. Otherwise, if a call on the original method is impermissible, then the yield call ends with no effect. On the other hand, when a call on the original method is impermissible, a force call will go ahead and execute the method body. It will then take necessary steps to repair the invariant by further modifying the state. Since invariants are application-specific, the repair is also application-specific. More importantly, it is up to the programmer to deem whether a specific repair procedure reflects the intended semantics of the object. As a trivial example, one can always repair the invariant by setting the post-state to the initial state. This will always satisfy the invariant, but is generally semantically an unacceptable solution. If there are multiple ways to repair the invariant, these can be offered in the form of multiple force methods.

For instance, consider a set S of pairs (x, y). The invariant is uniqueness over the the first element of pairs in S: $\forall (x_1, y_1), (x_2, y_2) \in S.\ (x_1 = x_2) \Rightarrow (x_1, y_1) = (x_2, y_2)$. Now consider the $\mathsf{add}(x, y)$ method that adds the pair (x, y) to S. This method clearly conflicts with itself for two calls that have the same first element. Suppose we replace add with add-yield and add-force. The former adds (x, y) to S, unless there already exists a pair whose first element is x; the latter adds (x, y) to S and additionally removes any pair whose first element is x. Note that for a pair (x_1, y_2), if there does not already exist a pair whose first element is x_1 in S, then the execution of add, add-yield, and add-force on (x_1, x_2) are indistinguishable.

We confirm these methods do not $\mathcal{P}_R$-conflict with themselves, nor each other. For two calls on add-yield with the same first element, only the pair of the first call ends up in S (first write wins). For two calls on add-force with the same first element, only the pair of the second call ends up in S (last write wins). For two calls on add-yield and add-force with the same first element, only the pair of add-force ends up in S (force wins).

7.3 Abstracting and Transforming Relational Schema into ORDTs

In this section, we show how relational database schema can be abstracted and transformed into ORDTs. A *relational database schema* is an object $\mathcal{S} = (\mathcal{D}, \mathcal{I}, \mathcal{M})$, where the database state $\mathcal{D}$ is a set of *tables*, the integrity property $\mathcal{I}$ is the conjunction of a set of SQL *constraints* on the tables, and the methods $\mathcal{M}$ are SQL commands. A table t is specified by its rows and columns. When referring to a row (column), we mean the set of values in that row (column). We assume there are no duplicate rows within a table, and a row and a column uniquely identify one value in the table which we call a cell (*i.e.*, first normal form).

We consider four types of SQL constraints. (a) The `NOT_NULL` constraint on a column c stipulates that the values of c are not the null value. (b) The `UNIQUE` constraint on a column c stipulates that the values of c are unique. (c) The `PRIMARY_KEY` (`PK`) constraint on a set of columns $\overline{c}$ of a table t stipulates that the rows of t have unique values over $\overline{c}$. A table has at most one primary key which is used to index and uniquely identify rows from other tables. (d) The `FOREIGN_KEY` (`FK`) constraint from a set of columns $\overline{c}$ of a table to the primary key $\overline{c'}$ of another table stipulates that every value in $\overline{c}$ is in $\overline{c'}$ as well. This constraint is commonly known as *referential integrity*.

Methods. We define the set of methods $\mathcal{M}$ available to the schema as SQL commands which can update and query the database. The client can add and delete row and columns, and update

existing values. (a) add-row(t, r): add row r to the table t. This method represents the SQL command INSERT INTO. (b) delete-row(t, ϕ): delete the rows that satisfy the condition ϕ from table t. This method represents the SQL command DELETE FROM WHERE. (c) update$(t, \phi, \langle\overline{c, v}\rangle)$: for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$. This method represents the SQL command UPDATE SET WHERE. (d) add-column(t, c, v): add column c with value v (defaults to null) to each row of table t. This method represents the SQL command ALTER TABLE ADD. (e) delete-column(t, c): delete column c (which is not in the primary key or a foreign key of t) from all rows of t. It represents the SQL command ALTER TABLE DROP COLUMN.

The client can add new tables and delete existing tables. (a) create-table$(t, \mathcal{I})$: add table t to the database and constraints $\mathcal{I}$ onto its columns. This method represents the SQL command CREATE TABLE. (b) delete-table(t): delete table t from the database. This method represents the SQL command DROP TABLE. Lastly, the client can execute arbitrary queries on the database that we capture as the query method. This method represents the SQL command SELECT FROM. Queries access, but do not mutate the database.

Conflict Relations. For brevity, as allowed by Lemma 4, we only concern ourselves with the permissibility graph of relational schema. Fig. 8 shows this graph. The cycles and loops represent two scenarios where permissible-conflicts happen: (c.1) cells of `UNIQUE` or `PK` columns are concurrently added or updated with the same value(s), or (c.2) a foreign key is added or updated to refer to some primary key while concurrently that particular primary key is deleted or updated to a different value. We refer to (c.1) conflicts as *uniqueness* conflicts and (c.2) conflicts as *referential integrity* conflicts. We note that (c.1) is an intra-table conflict and (c.2) is an inter-table conflict.

Consider the following schema:

t_1:

EmployeeID (PK)	Name	Department
001	Alice	Sales
002	Bob	Marketing

t_2:

EmployeeID (FK)	Salary
001	1000

An example of a (c.1) conflict is the calls add-row$(t_1, \langle$003, Carol, Sales$\rangle)$ and add-row$(t_1, \langle$003, Charlotte, Management$\rangle)$. Since the EmployeeID column has the PK constraint, its values must be unique; thus these two calls violate uniqueness. As an example of a (c.2) conflict, consider the calls add-row$(t_2, \langle 002, 300\rangle)$ and delete-row$(t_1, (\mathit{EmployeeID} = 002))$. The former adds a row to t_2 with a foreign key that refers to the primary key 002 of t_1, while the latter deletes the row with that specific primary key from t_1; thus, these two calls violate referential integrity. Notice that this happens independent of their order, hence both methods have an edge to the other.

Transformation and Abstraction. Now, we use a combination of abstractions and transformations to derive an object which is an ORDT that functions as a relational schema. We extend the abstraction and transformation examples from the previous section to general relational schema. First, we capture transformations that the SQL standard provides to resolve conflicts. In order to comply with referential integrity, we adopt the two referential actions: cascading or no-action. To preserve uniqueness, we similarly adopt yield and force semantics. In both cases, the transformed operation is semantically equivalent to the original operation when there is no conflict, but includes additional behavior when conflicts occur. We present a high-level sketch here; a comprehensive list of methods and analysis is available in the Appendix **??**. The conflict graph of the transformed and abstract object is shown in Fig. 9.

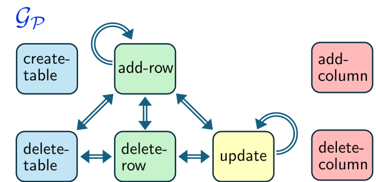

Fig. 8. Relational schema permissibility conflict graph.

Uniqueness. We resolve (c.1) conflicts by employing the yield and force transformations we saw in the previous subsection. More specifically, we transform the add-row method (by splitting it) into add-row-yield and add-row-force: (1) add-row-yield(t, r): add row r to table t, unless there exists another row in t that has the same values as r for a `UNIQUE` column or PK columns. (2) add-row-force(t, r): add row r to table t. Further, remove any rows in t that have the same values as r for a `UNIQUE` or PK columns. These methods do not $\mathcal{P}_R$-conflict with themselves or each other, using the same reasoning as before. The add-row-yield method does not add its specified row if it violates uniqueness. The add-row-force method adds its specified row, but also removes any rows that are not distinct from the specified row at `UNIQUE` or PK columns. Thus, uniqueness is preserved in the post-state. Since update also suffers from (c.1) conflicts, this transformation should be applied to update as well, giving us update-yield and update-force. (Note that this does not remove any edges incident to the update methods in $\mathcal{G}_{\mathcal{P}}$, as updates still suffer from (c.2) conflicts.)

Referential Integrity. In SQL, yield and force transformations are manifested as the CASCADE and NO ACTION referential actions. To resolve (c.2) conflicts, we transform the delete-row method with CASCADE and NO ACTION. In order to maintain referential integrity, when designating a foreign key in SQL, the user must specify the adaptation to the foreign key if the corresponding primary key is deleted (or updated). If a foreign key is set to CASCADE, then if a referent primary key is deleted/updated, all rows containing a foreign key that refers to that primary key are also deleted/updated. On the other hand, if the foreign key is set to NO ACTION, then the primary key cannot be deleted nor updated unless there are no rows whose foreign keys refer to that primary key. Accordingly, we specialize each method into two versions: cascade or no-action. (1) delete-row-cascade(t, ϕ): delete the rows that satisfy the condition ϕ from t. Further, if a deleted row contains a primary key k, then for all tables, delete all rows with a foreign key that refer to k. (2) delete-row-no-action(t, ϕ): delete the rows that satisfy the condition ϕ from t, unless a target row contains a primary key that is referred to by some foreign key.

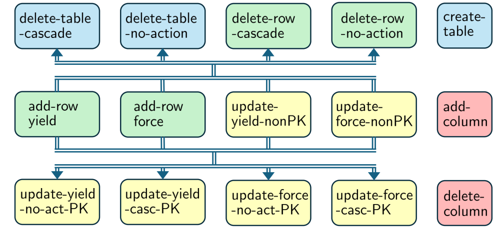

Fig. 9. Permissibility graph of the transformed and refined schema. Each of the (first four) methods in the middle row have an outgoing edge to each of the (first four) methods in the top and bottom rows.

We observe that although add-row-yield (and add-row-force) (still) $\mathcal{P}_R$-conflicts with delete-row-cascade (and delete-row-no-action), the opposite is not true. The delete method (either cascade or no-action) preserves referential integrity in the post-state. As before, since update (update-yield and update-force) has (c.2) conflicts, this transformation is applied to both update methods. This gives us cascade and no-action versions for each.

Primary Key Updates. On close inspection, each update method still conflicts with itself and every other update method. Consider two calls c_1 and c_2 on update-yield-cascade, for example. Let c_1 update some foreign key to refer to some primary key k, while c_2 updates k to a different value. Then, c_1 $\mathcal{P}_R$-conflicts with c_2; the cascade does not delete the new foreign key that is set to k afterwards. The solution to this problem is abstraction. More specifically, we split the update methods into two types, one for primary keys and one for everything else. Updates to primary keys always $\mathcal{P}_R$-commute with updates to non-primary key values, specifically foreign keys (but not the opposite). We split the update methods for PKs and non-PKs which are called when all, and respectively none, of the targeted columns are in the primary key of the table. Thus, we split each of the update variants further into two versions that end in PK and NonPK. The resulting object has a acyclic and loop-free permissibility graph and abstracts relational schema by construction.

Theorem 4. *There is an ORDT that abstracts the transformed relational schema.*

Proof. Immediate from Lemma 4, Fig. 9, and Definition 12. □

8 NP-Completeness

In § 5, we presented Frashokereti, an optimistic replication protocol for ORDTs, *i.e.*, objects with an acyclic conflict graph. As we saw in § 6, Frashokereti has two favorable properties: it is non-aborting and has polynomial-time local complexity. What about general objects; in particular, objects without acyclic conflict graphs? This set of objects is the complement of ORDTs, which we denote as $\overline{\text{ORDT}}$. Previous work presented optimistic protocols for $\overline{\text{ORDT}}$; however, they are aborting and have exponential-time local complexity. Two immediate questions are whether there are proper, non-aborting, optimistic protocols for $\overline{\text{ORDT}}$, and whether they can have polynomial-time local complexity. In this section, we first show that $\overline{\text{ORDT}}$ cannot have a proper, non-aborting, optimistic protocol. Any proper, optimistic protocol for $\overline{\text{ORDT}}$ must be *aborting*. Further, we formulate a decision version of the replication problem under aborting specifications and show it is NP-Complete.

Intuitively, a protocol cannot behave optimistically if there is a set of calls that cannot be reordered to preserve integrity. We formalize this intuition as follows.

Definition 14 (Tangled Set). *A set of calls s is tangled for a state σ if every call is permissible in σ, and the execution of every permutation of s from σ has at least one impermissible call.*

Lemma 5. *No object with a tangled set of calls for a state reachable from σ_0 permits a proper, non-aborting, optimistic protocol.*

Proof. Consider an object with a tangled set of calls s for state σ. Let some correct process p reach σ. By propagation, every correct process eventually executes the same set of calls as p; thereby, by convergence, eventually the (current) state of every correct process is σ. Further, by non-abortion, all calls up to σ are eventually committed, *i.e.*, cannot be reordered. At this point, let the calls of s be concurrently requested (infinitely often) by $|s|$ correct processes. By definition, every call in s is permissible at σ. By acceptance and optimism, each of these processes must eventually choose to execute their respective local request, without communicating with other processes. Then, let all of these calls, *i.e.*, calls of s, propagate between the processes. By propagation, the correct process p eventually executes every call in s. However, since s is tangled, p cannot arrange the calls in a way that every call is permissible, thus violating integrity. □

Corollary 5. *There is no proper, non-aborting, and optimistic protocol for general objects.*

Proof. Consider a bank account with non-negative balance as integrity. It provides withdraw and deposit methods. In the permissibility graph, the withdraw method has a loop. (This object is in $\overline{\text{ORDT}}$.) Two withdraw calls where their combined sum exceeds the balance form a tangled set. By Lemma 5, there is no proper, non-aborting, optimistic protocol for this object. □

According to Lemma 5, when an object has a tangled set of calls, any proper, optimistic protocol must abort a call to preserve integrity. For the remainder of this section, we consider protocols that may abort calls. Specifically, we weaken the non-abortion property to allow for the abortion of tentative calls under certain conditions. This must be done carefully to prevent the trivial solution of aborting every call. Thus, we allow a process to abort a call only when necessary, *i.e.*, encounters a tangled set. (In fact, the process *must* abort one of the calls.)

DEFINITION 15 (AS-NECESSARY-ABORTING). *If a correct process executes a call c, then it eventually commits c, unless c is a member of a tangled set s for the committed state, and all calls in s other than c eventually commit.*

This property stipulates that a call can only be aborted only if it is part of a tangled set. We now consider replacing the non-aborting property with the as-necessary-aborting property: we consider proper, as-necessary-aborting, optimistic protocols. Any such protocol must be able to determine whether a given set of calls is tangled for a given state. Specifically, if the set is tangled, then the protocol must choose to abort a call, since by definition, at least one call of every tangled set is impermissible. On the other hand, if the set is not tangled, the protocol must execute every call in the set, since the as-necessary-abortion property allows for the abortion of a call only if it is a member of a tangled set.

We abstractly capture this problem, determining whether a given set of calls is tangled for a given state, as the co-Tangled decision problem: given an object $O = \langle \sigma_0, \mathcal{I}, \mathcal{M} \rangle$ and a set of calls s on O where every call $c \in s$ is permissible in σ_0, the objective is to decide if s is *not* tangled for σ_0, *i.e.*, there exists a permutation π of s such that every call in π is permissible upon executing π on σ_0. In fact, such a protocol would need to repeatedly decide instances of co-Tangled throughout its execution (on different starting states σ_0). We prove co-Tangled is NP-Complete. Thus, any proper, as-necessary-aborting, optimistic protocol has non-polynomial time complexity (unless P = NP).

THEOREM 6. co-Tangled *is* NP*-Complete.*

The full reduction and proof can be found in the Appendix **??**; we provide a sketch here. The reduction is from the 3COLORING problem: given a graph $G = (V, E)$, the objective is to decide if there exists a coloring $F : V \longrightarrow \{\mathsf{red}, \mathsf{blue}, \mathsf{green}\}$ such that no two adjacent vertices share a color. Given an instance of 3COLORING, we reduce it to an instance of co-Tangled as follows: the object state is graph G; the initial state σ_0 has every vertex colorless; the invariant $\mathcal{I}$ is that no two adjacent vertices share a color; there are four methods in $\mathcal{M}$: red, blue, green, which set the input vertex to the corresponding color, and clear, which removes the color of every vertex only if every vertex has a color, and does nothing otherwise. Lastly, let set $s = \{\mathsf{red}(v_i), \mathsf{blue}(v_i), \mathsf{green}(v_i), \mathsf{clear}, \mathsf{clear}\}$, for every vertex v_i in G.

Next, we show G has a 3-coloring F iff s is not tangled for σ_0. ($\Rightarrow$) We use F to construct permutation π of s where every call in π is permissible. The idea is to color the object graph three times, using clear in between each time. That is, place the calls as follows: calls corresponding to F first, a clear call, calls corresponding to a rotation of F, the second clear, then the remaining calls. ($\Leftarrow$) We use the permissible permutation π to construct a coloring F. We claim that for some post-state in π, every vertex in the graph has a color. Since every call in π is permissible, then no adjacent vertices share a color. We define F by observing the color of the vertices in that post-state.

9 Experimental Results

We implemented FRASHOKERETI in C++14. In this section, we present our experimental results for FRASHOKERETI. We first compare FRASHOKERETI with conflict-free objects, *i.e.*, objects with no conflicts. The experiments exhibit that FRASHOKERETI has comparable results to CRDTs [60]: on average, FRASHOKERETI's latency and throughput are worse but within 1.9% and 4.7% margin of CRDTs. We then use FRASHOKERETI to replicate object use-cases with conflicting methods including the Project, Movie and Courseware usecases that we adopted from previous work [32, 43]. We adopt ECRO as the baseline as it is the most recent optimistic replication system. We compare FRASHOKERETI with ECRO for both latency and throughput. Experiments show that as the number of replicas increases, FRASHOKERETI scales: its throughput increases while its latency slightly

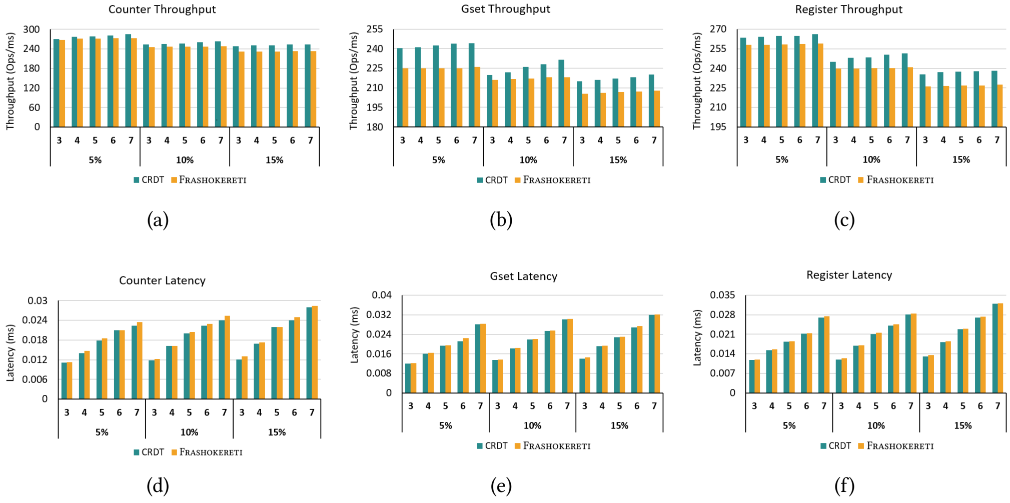

Fig. 10. FRASHOKERETI vs. CRDTs on conflict-free use-cases. (The same plots are magnified in the appendix ??.)

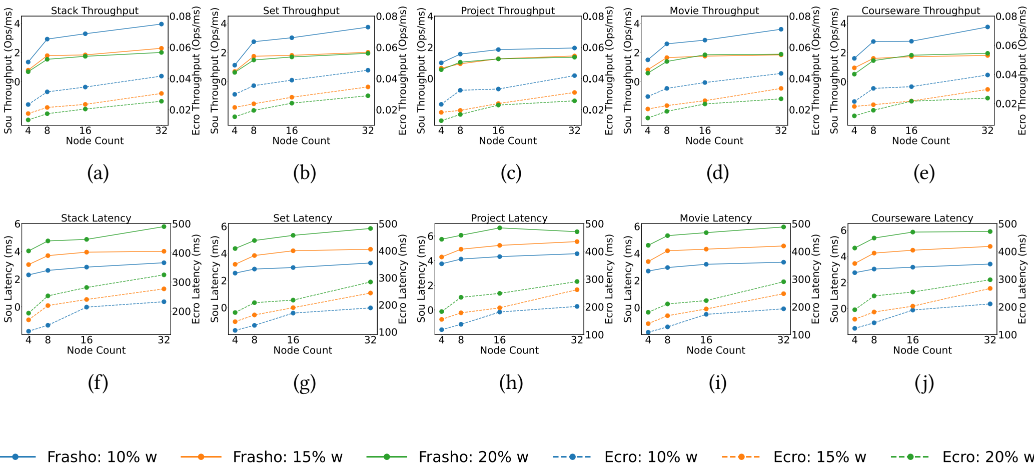

Fig. 11. Optimistic replication of conflicting use-cases with FRASHOKERETI. Note that the left y-axis is for FRASHOKERETI and the right y-axis is for ECRO. (The same plots are magnified in the appendix ??.)

increases. Additionally, FRASHOKERETI provides significantly higher throughput and lower latency compared to ECRO. On average, FRASHOKERETI's latency is 47.77x lower and FRASHOKERETI's throughput is 66.45x higher than ECRO. The throughput is calculated by dividing the total number of calls by the time it takes for all calls to be replicated on all replicas. The latency (or response time) is calculated as the average duration between the request and the response over all the calls.

Questions. The experiments seek to answer the following questions: Is FRASHOKERETI competitive to CRDTs? Is FRASHOKERETI's throughput scalable as the number of replicas increase? How does increasing the number of replicas affect FRASHOKERETI's latency? How does FRASHOKERETI compare to other optimistic replication protocols like ECRO? Can FRASHOKERETI gracefully tolerate failures?

Conflict-Free Objects. *Platform and Setup.* To evaluate CRDTs, we deployed our optimistically replicated objects on NSF HPRC ACES cluster, using the Sapphire Rapids nodes, which run Intel Xeon 8468 (Sapphire Rapids) with 512 GB DDR5 memory, networked with NVIDIA Mellanox NDR200 (200Gbps), and connected with NDR400 network switches (400Gbps). We performed the experiments on a 7-node cluster. We compared FRASHOKERETI on three CRDTs: Counter, Grow-Only-Set (GSet), and Last-Write-Wins register (LWW). In each run, we issued a workload of 12k operations. We experimented with 5%, 10%, and 15% write (versus reads). We randomly generated method calls and uniformly distributed update calls over methods.

Evaluation. Fig. 10 presents the throughput and latency results for CRDTs and FRASHOKERETI with both increasing number of replicas, and varying workloads. As the number of replicas increase, the throughput goes up, showing that FRASHOKERETI is scalable. Further, as the number of replicas increases, the latency goes up, which is expected since more packets need to be sent. We observe that FRASHOKERETI stays in the same ballpark as the CRDTs within a margin of 4.7% for throughput, and 1.9% for latency. In order to handle conflicting calls, FRASHOKERETI maintains a sequence of tentative calls that CRDTs don't. However, for conflict-free objects, FRASHOKERETI can tailor a particular path that skips maintaining the tentative calls. Then, our additional experiments showed that there will be minor difference in the performance of CRDTs and FRASHOKERETI. Since the difference is only statistical sampling error, we avoid presenting plots for it.

Objects With Conflicts. *Platform and Setup.* To evaluate conflicting use cases and to scale our experiments up to 32 nodes, we deployed FRASHOKERETI on a paid Google Cloud Platform service using e2-highcpu-8 instances with AMD EPYC 7B12 processors, 8 GB of ECC RAM, and virtual NICs supporting up to 16 Gbps bandwidth. It runs Red Hat Enterprise 8 with gcc 8.5. Similar to the previous experiment, in each run, we issued a workload of 12k operations, with 5%, 10%, and 15% write (versus reads), and randomly generated calls uniformly distributed over methods.

We evaluated FRASHOKERETI on 5 conflicting use-cases: Set, Stack, Project, Movie, and Courseware. Set has two methods: add and remove. Stack has two methods: push and pop. Project has five methods: add-project, delete-project, works-on, add-employee, and delete-employee. Movie has four methods: add-movie, delete-movie, add-customer, and delete-customer. Finally, Courseware has five methods: add-course, delete-course, enroll, add-student, and delete-student.

Evaluation. Fig. 11 presents the throughput and latency FRASHOKERETI and ECRO for the above use cases, as we change the number of nodes and read versus write ratio in the workload. We first tried to use an implementation from ECRO's repository, but only JAR files and not the source codes were publicly available. Thus, we re-implemented the ECRO protocol in C++. As the number of replicas increases, FRASHOKERETI's throughput scales across use cases and workloads. At the same time, its latency only slightly increases. In comparison with ECRO, optimistic replication provides on average 66.45x higher throughput, and 47.77x lower latency. Both FRASHOKERETI and ECRO are optimistic protocols. However, FRASHOKERETI executes calls by efficient handlers with polynomial time complexity, which enhances its performance. In contrast, ECRO maintains a graph during runtime which can grow large, and further performs an exponential time computation on the graph to decide which calls to abort, and how to order the rest.

Workload Ratios. In this experiment, we measure performance for different workloads: 25, 50, 75 and 100% writes. We replicated the five objects we considered above on 4 GCP ec2-highcpu-8 nodes, and issued 12k operations. Figure 12(a) and 12(b) show that as the write percentage increases,

the throughput decreases, and the latency increases. This trend is expected as more writes generally incur more conflicts.

Larger Benchmarks. In addition, we evaluate FRASHOKERETI on two larger benchmarks: YCSB [24], and extended courseware. We extended the courseware example into a larger relational schema with 6 relations and 20 methods. We make this benchmark and its coordination analysis available. It has four independent relations STUDENT, INSTRUCTOR, COURSE, and DEPARTMENT, and two referencing relations ENROLLMENT(course, student) and TEACHING(course, instructor). In addition to referential integrity constraints, the course name has a uniqueness constraint. The benchmark methods are explained in the appendix. We continuously issue operations, and measure the throughput and latency in 50 seconds after 2 minutes of warm up. The throughput and latency of FRASHOKERETI for YCSB are presented in Figure 12(c), and 12(d), and for extended courseware are presented in Figure 12(e), and 12(f). As expected, as the number of nodes increases, the latency generally increases, and the throughput increases in YCSB. Further, the workloads with higher read loads gain higher performance.

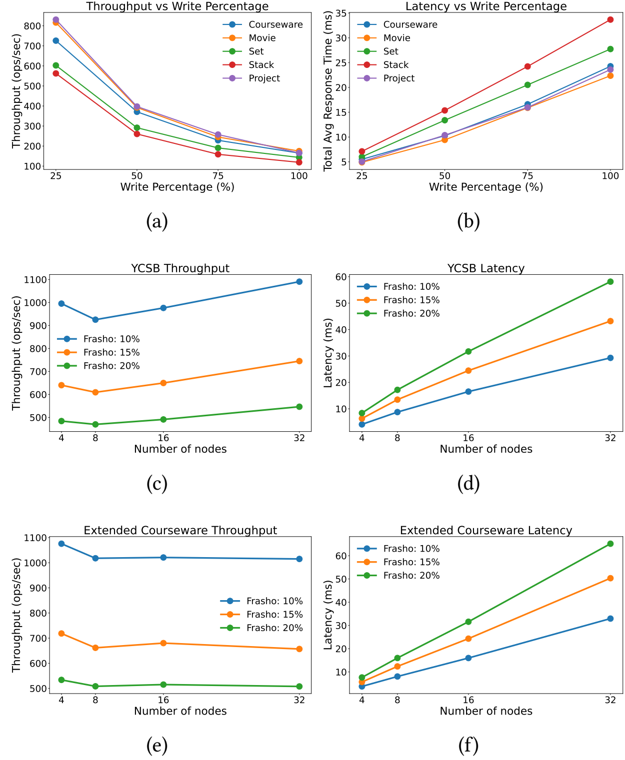

Fig. 12. (a) and (b) Different workload ratios. (c) and (d) YCSB. (e) and (f) Extended courseware.

Fault-Tolerance. *Platform and Setup.* The platform is eight 4-core machines (Google Cloud e2-highcpu-4 instances). We run the stack use-case for 3 minutes with a 50% write ratio, and inject 0 to 2 node failures.

Evaluation. Fig. 13 shows the throughput over time under 0, 1 and 2 failing processes. We observe that FRASHOKERETI can gracefully tolerate failures and continue the execution of requests. Further, as the number of nodes is decreased, the stabilization cost decreases and the throughput increases.

10 Related Works

Conflict-free Replicated Data Types (CRDTs) [2, 60, 61] characterize a class of objects that converge without coordination. They have been a subject of specification [16, 17, 44, 67], verification [13, 33, 48, 53], synthesis [3, 15, 36], and incorporation into programming models [47]. They were later followed by more expressive convergent data types such as cloud, mergeable types [20, 21, 34, 39] that reconcile calls into a convergent state. Compared to convergent data types, ORDTs supports

a larger class of objects: objects with acyclic, rather than empty, conflict graphs, and further accommodates application invariants.

Unfortunately, CRDTs require every pair of calls on the object to be commutative, limiting their applicability. Further, they are primarily concerned with convergence, but not integrity [7]. This inspired the development of hybrid solutions, which fundamentally behave optimistically when possible and default to coordination when necessary. Works like IPA [8], RedBlue [40–42], Quelea [55], Indigo [9], CISE [29] Hamsaz [32], Hampa [43], and LoRe [30] statically analyze the object for invariant conflicts. IPA suggests modifications to the methods to avoid conflicts at runtime. Indigo offers the choice between avoidance, *e.g.*, reservations, and repair subroutines to patch the invariant after violations. RedBlue, Hamsaz, and ECROs categorize methods that require coordination and enforce strong consistency only on those calls. Quelea takes operation contracts, and determines the weakest consistency model that preserve invariants for each operation. CISE provides a proof rule to verify integrity of a replicated object under chosen consistency conditions for its methods. LoRe injects coordination into reactive local-first programs to provably preserve user-defined invariants. Escrow mechanisms and its generalizations [9, 11, 31, 35, 45, 46, 51, 54, 56, 58, 68–70] are also hybrid in principle; they perform operations locally when they do not risk integrity, but ultimately resort to global synchronization for redistribution and reconfiguration. In contrast, the Frashokereti protocol operates on ORDTs in a fully optimistic manner, never resorting to coordination.

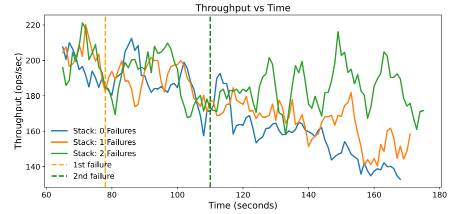

Fig. 13. Fault tolerance. Throughput over time. The vertical lines show the time of fault injection.

Motivated by coordination-freedom of CRDTs to preserve convergence, $\mathcal{I}$-confluence [6] and Soteria [49] capture the conditions under which replicated objects can preserve integrity without coordination. However, similar to CRDTs, these classes don't include general objects. ECRO [23] presents a mostly-optimistic protocol for general objects. It statically analyzes methods to determine those that require coordination, and performs coordination for them. It optimistically executes the other methods, and dynamically resolves conflicts by searching for an ordering that preserves the integrity for every call. However, this comes at the expense of (1) potentially aborting calls and (2) solving an exponential-time graph problem (per call). On the other hand, Frashokereti avoids both: it never aborts calls and has polynomial-time local complexity.

In order to preserve convergence and integrity, several works proposed modifications to conflicting methods. Operational transformations [25, 63] modify the parameters of concurrent conflicting calls, but not the method bodies. As mentioned, IPA [8] suggests modifications to methods when its static analysis discovers conflicts. No-Op [14] considers priorities for methods, and executes a low priority call as a no-op when a conflict occurs. Instead, Bayou [65] and Indigo [9] attach additional behavior to calls to repair the invariant. Similarly, this paper presents the *yield* semantics to neutralize the call, and the *force* semantics to repair the integrity. Further, this paper introduces the refinement technique: splitting methods into refined methods that can reduce conflicts.

11 Conclusion

This paper proves that optimistic replication for general objects is aborting and NP-Complete. It characterizes ORDTs, objects that can be optimistically replicated with convergence and integrity, and particularly without aborting calls. It shows that ORDTs subsume CRDTs and transformed relational schema. Further, it presents an efficient and provably sound replication protocol Frashokereti for ORDTs.

12 Data-Availability Statement

We will submit all the development framework including the code, use-cases and experiments to the artifact evaluation track, and further we will release them as open source software at the time the paper is presented.

Acknowledgments

This project has been supported by NSF 2437238, DARPA D22AP00146-02.

References

[1] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design. *Computer* 45, 2 (2012), 6 pages. doi:10.1109/MC.2012.33

[2] Paulo Sérgio Almeida. 2024. Approaches to Conflict-free Replicated Data Types. *Comput. Surveys* 57, 2 (2024), 1–36. doi:10.1145/3695249

[3] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2017. Blazes: Coordination Analysis and Placement for Distributed Programs. *ACM Trans. Database Syst.* 42, 4, Article 23 (October 2017), 31 pages. doi:10.1145/3110214

[4] 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.

[5] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2021. Specification and Space Complexity of Collaborative Text Editing. *Theor. Comput. Sci.* 855 (2021), 141–160. doi:10.1016/J.TCS.2020.11.046

[6] 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 (November 2014), 185–196. doi:10.14778/2735508.2735509

[7] 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. doi:10.1145/2723372.2737784

[8] 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.]). doi:10.14778/3297753.3297760

[9] 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.

[10] 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 (January 2015), 121–125. doi:10.1145/2723872.2723889

[11] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS).* IEEE, 31–36. doi:10.1109/SRDS.2015.32

[12] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. 2017. Pure Operation-Based Replicated Data Types. arXiv:1710.04469 [cs.DC]

[13] Ranadeep Biswas, Michael Emmi, and Constantin Enea. 2019. On the Complexity of Checking Consistency for Replicated Data Types. In *International Conference on Computer Aided Verification.* Springer, 324–343. doi:/10.1007/978-3-030-25543-5_19

[14] Dina Borrego, Nuno Preguiça, Elisa Gonzalez Boix, and Carla Ferreira. 2025. Ensuring Convergence and Invariants Without Coordination. In *39th European Conference on Object-Oriented Programming (ECOOP 2025).* Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 4–1. doi:10.4230/LIPIcs.ECOOP.2025.4

[15] Dina Borrego, Afonso Vilalonga, Henrique Domingos, Nuno Preguiça, Elisa Gonzalez Boix, and Carla Ferreira. 2025. ReDunT: Automatically Deriving Redundancy Relations for Pure Op-Based CRDTs. In *Proceedings of the 12th Workshop on Principles and Practice of Consistency for Distributed Data.* 38–44. doi:10.1145/3721473.3722145

[16] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. 2014. Verifying eventual consistency of optimistic replication systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14).* Association for Computing Machinery, New York, NY, USA, 285–296. doi:10.1145/2535838.2535877

[17] Ahmed Bouajjani, Constantin Enea, Madhavan Mukund, Gautham Shenoy R., and S. P. Suresh. 2020. Formalizing and Checking Multilevel Consistency. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-030-39322-9_18

[18] Eric Brewer. 2012. CAP twelve years later: How the" rules" have changed. *Computer* 45, 2 (2012), 23–29. doi:10.1109/MC.2012.37

[19] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7. Portland, OR, 343477–343502. doi:10.1145/343477.343502

[20] 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. doi:10.1145/1869459.1869515

[21] 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. doi:10.1007/978-3-642-31057-7_14

[22] 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 (August 2013), 22 pages. doi:10.1145/2491245

[23] 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. doi:10.1145/3485484

[24] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1085–1100. doi:10.1145/3035918.3064033

[25] Clarence A Ellis and Simon J Gibbs. 1989. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. 399–407. doi:10.1145/66926.66963

[26] Colin J Fidge. 1987. Timestamps in message-passing systems that preserve the partial ordering. (1987).

[27] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 9 pages. doi:10.1145/564585.564601

[28] Seth Gilbert and Nancy A. Lynch. 2012. Perspectives on the CAP Theorem. *IEEE Computer* 45, 2 (2012), 30–36. doi:10.1109/MC.2011.389

[29] 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. doi:10.1145/2837614.2837625

[30] Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. 2024. LoRe: A programming model for verifiably safe local-first software. *ACM Transactions on Programming Languages and Systems* 46, 1 (2024), 1–26. doi:10.1145/3633769

[31] 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. doi:10.1145/2987550.2987559

[32] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication coordination analysis and synthesis. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–32. doi:10.1145/3290387

[33] 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. doi:10.1145/3276534

[34] 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. doi:10.1145/3360580

[35] Narayanan Krishnakumar and Arthur J Bernstein. 1992. High Throughput Escrow Algorithms for Replicated Databases. In *VLDB*, Vol. 1992. 175–186.

[36] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M Hellerstein. 2022. Katara: Synthesizing CRDTs with verified lifting. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1349–1377. doi:10.1145/3563336

[37] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. doi:10.1145/359545.359563

[38] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998). doi:10.1145/279227.279229

[39] 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. doi:10.1145/3341710

[40] 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.

[41] 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. doi:10.1145/2745947.2745955

[42] 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

[43] Xiao Li, Farzin Houshmand, and Mohsen Lesani. 2020. Hampa: Solver-Aided Recency-Aware Replication. In *International Conference on Computer Aided Verification*. Springer, 324–349. doi:10.1007/978-3-030-53288-8_16

[44] 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. doi:10.1145/3453483.3454067

[45] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. 2014. Warranties for Faster Strong Consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, Berkeley, CA, USA, 503–517. http://dl.acm.org/citation.cfm?id=2616448.2616495

[46] Tom Magrino, Jed Liu, Nate Foster, Johannes Gehrke, and Andrew C Myers. 2019. Efficient, consistent distributed computation with predictive treaties. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16. doi:10.1145/3302424.3303987

[47] 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. doi:10.1145/3360570

[48] Kartik Nagar and Suresh Jagannathan. 2019. Automated Parameterized Verification of CRDTs. In *International Conference on Computer Aided Verification*. Springer, 459–477. doi:10.1007/978-3-030-25543-5_26

[49] Sreeja Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the safety of highly-available distributed objects. In *ESOP 2020-29th European Symposium on Programming*. doi:10.1007/978-3-030-44914-8_20

[50] 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. doi:10.1145/62546.62549

[51] Patrick E O'Neil. 1986. The escrow transactional method. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 405–430. doi:10.1145/7239.7265

[52] 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.

[53] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. 2024. VeriFx: Correct Replicated Data Types for the Masses. arXiv:2207.02502 [cs.PL] https://arxiv.org/abs/2207.02502

[54] Nuno Preguiça, J Legatheaux Martins, Miguel Cunha, and Henrique Domingos. 2003. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st international conference on Mobile systems, applications and services*. 43–56.

[55] Kia Rahmani, Gowtham Kaki, and Suresh Jagannathan. 2018. Fine-grained distributed consistency guarantees with effect orchestration. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data* (Porto, Portugal) *(PaPoC '18)*. Association for Computing Machinery, New York, NY, USA, Article 6, 5 pages. doi:10.1145/3194261.3194267

[56] Andreas Reuter. 1982. Concurrency on high-traffic data elements. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 83–92. doi:10.1145/588111.588126

[57] 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. doi:10.1016/j.jpdc.2010.12.006

[58] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. 2015. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 1311–1326. doi:10.1145/2723372.2723720

[59] Yasushi Saito and Marc Shapiro. 2005. Optimistic replication. *ACM Computing Surveys (CSUR)* 37, 1 (2005), 42–81. doi:10.1145/1057977.1057980

[60] 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.

[61] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400. doi:10.1007/978-3-642-24550-3_29

[62] 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. doi:10.1145/2737924.2737981
[63] Chengzheng Sun and Clarence Ellis. 1998. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 59–68. doi:10.1145/289444.289469
[64] Douglas B Terry, Marvin Theimer, Karin Petersen, and Mike Spreitzer. 2000. An examination of conflicts in a weakly-consistent, replicated application. *Personal communication* (2000).
[65] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 172–182. doi:10.1145/224057.224070
[66] Marko Vukolic. 2016. Eventually Returning to Strong Consistency. *IEEE Data Eng. Bull.* 39, 1 (2016), 39–44.
[67] 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. doi:10.1145/3314221.3314617
[68] Michael Whittaker and Joseph M Hellerstein. 2018. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment* 12, 1 (2018), 14–27. doi:10.14778/3275536.3275538
[69] Michael Whittaker and Joseph M Hellerstein. 2020. Checking invariant confluence, in whole or in parts. *ACM SIGMOD Record* 49, 1 (2020), 7–14. doi:10.1145/3422648.3422651
[70] Haifeng Yu and Amin Vahdat. 2000. Efficient numerical error bounding for replicated network services. In *VLDB*. Citeseer, 123–133.

Received 2025-10-10; accepted 2026-02-17