Hamsaz: Replication Coordination Analysis and Synthesis*

FARZIN HOUSHMAND, University of California, Riverside, USA
MOHSEN LESANI, University of California, Riverside, USA

Distributed system replication is widely used as a means of fault-tolerance and scalability. However, it provides a spectrum of consistency choices that impose a dilemma for clients between correctness, responsiveness and availability. Given a sequential object and its integrity properties, we automatically synthesize a replicated object that guarantees state integrity and convergence and avoids unnecessary coordination. Our approach is based on a novel sufficient condition for integrity and convergence called well-coordination that requires certain orders between conflicting and dependent operations. We statically analyze the given sequential object to decide its conflicting and dependent methods and use this information to avoid coordination. We present novel coordination protocols that are parametric in terms of the analysis results and provide the well-coordination requirements. We implemented a tool called Hamsaz that can automatically analyze the given object, instantiate the protocols and synthesize replicated objects. We have applied Hamsaz to a suite of use-cases and synthesized replicated objects that are significantly more responsive than the strongly consistent baseline.

CCS Concepts: • **Theory of computation** → **Invariants**; **Program analysis**; • **Software and its engineering** → **Distributed programming languages**; *Distributed systems organizing principles*;

Additional Key Words and Phrases: Well-Coordination, Distributed Systems, Invariant-Preserving, Consistency, Program Synthesis

ACM Reference Format:
Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *Proc. ACM Program. Lang.* 3, POPL, Article 74 (January 2019), 32 pages. https://doi.org/10.1145/3290387

1 INTRODUCTION

Distributed system replication [Belaramani et al. 2006; Birman 1985; Ladin et al. 1992; Petersen et al. 1997] is an often-used mechanism to achieve fault-tolerance and scalability. Embedded control systems replicate controllers [Madhusudan and Thiagarajan 2001] to tolerate faults, online services rely on geo-replicated data stores [Cooper et al. 2008; Corbett et al. 2013; DeCandia et al. 2007; Li et al. 2012; Lloyd et al. 2011, 2013; Sovran et al. 2011] to manage the ever-growing amount of data and hand-held devices replicate data for off-line use. There has been a known dilemma [Abadi 2012; Fischer et al. 1985; Gilbert and Lynch 2002, 2012] between strong and weak consistency of replicated objects. Strongly consistent replication (via Viewstamp [Oki and Liskov 1988], Paxos [Lamport 1998] and Raft [Ongaro and Ousterhout 2014] protocols) guarantees the same total order of operations across all replicas. Therefore, if an operation is checked to preserve the integrity properties [Bailis et al. 2015] at a replica, it will certainly preserve them in the other replicas as well. Further, replicas

*This work was partially funded by the National Science Foundation grant, *CRII: SHF: Certified Byzantine Fault-tolerant Systems* (1657204).

Authors' addresses: Farzin Houshmand, University of California, Riverside, USA, fhous001@ucr.edu; Mohsen Lesani, University of California, Riverside, USA, lesani@cs.ucr.edu.

2475-1421/2019/1-ART74
https://doi.org/10.1145/3290387

converge as a result of the same sequence of operations. Therefore, the correctness of replicated execution simply reduces to the correctness of sequential execution. However, synchronisation protocols that provide strong consistency need consensus between replicas and, hence, may not be responsive and even available during network failures or offline use. Although optimized protocols can emerge [Corbett et al. 2013; Jin et al. 2018], the strong semantics prevents their availability for offline use. On the other hand, weak consistency notions can be provided with availability and responsiveness but without the same total order of operations across replicas. Many consistency weak notions dubbed eventual consistency [Bouajjani et al. 2014; Burckhardt et al. 2014; Clancy and Miller 2017; Emmi and Enea 2018; Shapiro et al. 2011; Vogels 2008] simply broadcast the operations that may be arbitrarily reordered. Likewise, causal consistency [Ahamad et al. 1995; Birman 1985; Lamport 1978] preserves only the causal order between operations. Unfortunately, the absence of the total order can lead to violation of integrity properties.

However, weak notions can be enough for certain operations to preserve the integrity properties. For example, consider a bank account object with the integrity property that its balance is non-negative. The deposit operation can be executed without any coordination as it cannot make the balance negative. However, a withdraw operation has to synchronize with other withdraw operations to avoid overdrafts. In addition, consistent execution of a withdraw operation may be dependent on the preceding deposit operations in the originating replica. Therefore, the withdraw operation needs both a total order with respect to other withdraw operations and a causal order with respect to preceding deposit operations. We observe that operations have distinct coordination requirements with respect to each other. It is unintuitive for end-users to specify the right consistency requirement for each operation. Requesting too much may degrade performance, and requesting too little may compromise correctness. Thus, users either resort to the blanket strong consistency for all operations or ignore the problem and use a default notion of weak consistency.

Previous work recognized the problem, proposed hybrid models and took significant steps towards helping the user with consistency choices [Balegas et al. 2015a; Gotsman et al. 2016; Li et al. 2014, 2015; Sivaramakrishnan et al. 2015; Terry et al. 2013] to avoid coordination [Bailis et al. 2014; Roy et al. 2015]. They proposed proof techniques to verify the sufficiency of user-specified consistency choices [Gotsman et al. 2016], or require user annotations to identify consistency choices and do not guarantee convergence [Balegas et al. 2015a]. Further, many approaches [Balegas et al. 2015a; Gotsman et al. 2016; Li et al. 2014] are crucially dependent on causal consistency as the weakest possible notion while others have established the scalability limitations of causal consistency [Bailis et al. 2012a]. We will further survey related works in § 9. Given a sequential object with its integrity properties, our goal is to automatically synthesize a correct-by-construction replicated object that guarantees integrity and convergence and avoids unnecessary coordination: synchronization and tracking dependency between operations. Further, our approach supports notions weaker than causal consistency; it builds upon eventual, causal and strong notions.

We present a static analysis and protocol co-design. The core of our approach is a novel sufficient condition called well-coordination for integrity and convergence of replicated objects. We define notions of conflicting and dependent pairs of methods. Well-coordination requires synchronization between conflicting and causality between dependent operations. We statically analyze the given sequential object and its integrity property, and infer the pairs of conflicting methods (represented as the conflict graph) and dependent methods. We present two novel distributed protocols that provide the well-coordination requirements. The protocols are parametric for the analysis results. We present a non-blocking synchronization protocol based on a novel variant of the total-order-broadcast protocol. The protocol parameters are decided by a reduction of the minimum synchronization problem to the maximal clique problem on the conflict graph. We also present a synchronization protocol that is blocking but allows some of the conflicting methods

```
Class Courseware
    let Student := Set ⟨sid: SId⟩ in
    let Course := Set ⟨cid: CId⟩ in
    let Enrolment :=
        Set ⟨esid: SId, ecid: CId⟩ in
    $\Sigma$ := Student × Course × Enrolment
    $\mathcal{I}$ := $\lambda\ \langle ss, cs, es \rangle$.
        refIntegrity($es$, esid, $ss$, sid) ∧
        refIntegrity($es$, ecid, $cs$, cid)
    register($s$) := $\lambda\ \langle ss, cs, es \rangle$.
        $\langle \mathbb{T},\quad \langle ss \cup \{s\}, cs, es \rangle,\quad \bot \rangle$
    addCourse($c$) := $\lambda\ \langle ss, cs, es \rangle$.
        $\langle \mathbb{T},\quad \langle ss, cs \cup \{c\}, es \rangle,\quad \bot \rangle$
    enroll($s, c$) := $\lambda\ \langle ss, cs, es \rangle$.
        $\langle \mathbb{T},\quad \langle ss, cs, es \cup \{(s, c)\} \rangle,\quad \bot \rangle$
    deleteCourse($c$) := $\lambda\ \langle ss, cs, es \rangle$.
        $\langle \mathbb{T},\quad \langle ss, cs \setminus \{c\}, es \rangle,\quad \bot \rangle$
    query := $\lambda\ \sigma.\ \langle \mathbb{T},\quad \sigma,\quad \sigma \rangle$
```

(a) User Specification

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	×	✓
e	✓	✓	✓	✓	✓
d	✓	×	✓	✓	✓
q	✓	✓	✓	✓	✓

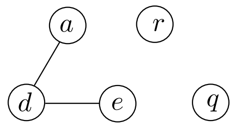
(b) $\mathcal{S}$-commute

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	✓	✓	✓	×	✓
d	✓	✓	×	✓	✓
q	✓	✓	✓	✓	✓

(c) $\mathcal{P}$-concur

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	×	✓
e	✓	✓	✓	×	✓
d	✓	×	×	✓	✓
q	✓	✓	✓	✓	✓

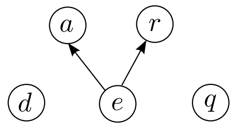
(d) Concur



(e) Conflict Graph $G_{\bowtie}$

	r	a	e	d	q
r	✓	✓	✓	✓	✓
a	✓	✓	✓	✓	✓
e	×	×	✓	✓	✓
d	✓	✓	✓	✓	✓
q	✓	✓	✓	✓	✓

(f) Independent



(g) Dependency Graph

Fig. 1. Courseware Use-case. $\text{refIntegrity}(R, f, R', f') := \forall r.\ r \in R \rightarrow \exists r'.\ r' \in R' \land f(r) = f'(r')$

to execute without synchronization. The protocol parameters are decided by a reduction of the minimum synchronization problem to the vertex cover problem on the conflict graph.

We present a tool called Hamsaz that given an object definition, uses off-the-shelf SMT solvers to decide the pairs of conflicting and dependent methods. It then uses the analysis results to avoid coordination and instantiate the protocols to synthesize replicated objects. We successfully synthesized replicated objects for a suite of use-cases that we have adopted from the previous works including CRDTs, bank account, auction, courseware, payroll and tournament. Experiments show that compared to the strongly consistent baseline, the synthesized replicated objects are significantly more responsive.

In the rest of the paper, we first present an overview in § 2. We define the well-coordination condition and prove its sufficiency for correctness in § 3. We present the static analysis and apply it to use-cases in § 4 and § 5. We then, present the protocols in § 6. The implementation and evaluation are presented in § 7 and 8 before we conclude with related works and final remarks in § 9 and 10.

2 OVERVIEW

In this section, we illustrate the coordination analysis and synthesis with examples.

Object Replication. We define an object as a record $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ that includes the state type Σ, an *invariant* $\mathcal{I}$ that is a predicate on the state, and a set of methods $\mathcal{M}$. Fig. 1.(a) represents the courseware object that we have adopted from [Gotsman et al. 2016]. The state type Σ is the tuple of three relations for students *ss*, courses *cs* and enrolments *es* of students in courses. A relation is a set of records of fields. The student and course relations *ss* and *cs* are simply a set of records of one field, student identifiers sid and course identifiers cid respectively. The enrolment relation *es* is a set of records of two fields: the student identifier esid and the course identifier ecid, that are foreign keys from the other two relations. The desired invariant $\mathcal{I}$ for the courseware object is the referential integrity of the two foreign keys of the enrolment relation *es*. Every student identifier esid in the enrolment relation *es* must refer to an existing student identifier sid in the student relation *ss*. The condition for course identifiers is similar. We represent referential integrity properties using the refIntegrity predicate (defined in the caption). For example, refIntegrity$(es, \mathsf{esid}, ss, \mathsf{sid})$ states that for every record r in the relation es, there exists a record r' in the relation ss such that esid of r is equal to sid of r' that is $\mathsf{esid}(r) = \mathsf{sid}(r')$ where the field names esid and sid are used as functions on the corresponding records. Methods represent transactions on the object state. A method is a function m from the method parameter(s) and the pre-state σ to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$, where guard is the boolean precondition of the method, update is the post-state, and retv is the return value. The courseware object has five methods: register to register a student, addCourse to add a course, enroll to enroll a student in a course, deleteCourse to delete a course and query to obtain the current state of the object. The guard of a method captures the semantic preconditions of the method and *not* the conditions that preserve the invariant. (We present the conditions that preserve the invariant in § 3.) For simplicity, the guards in this example are all $\mathbb{T}$. (A guard for the deleteCourse method could be that the input course should exist in the course relation to be deleted.) All but the query method return no value $\bot$. A method call c is the application of a method to its arguments i.e. a function from the pre-state to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$.

Given the definition of a sequential object, the goal is to automatically synthesize a replicated object. The state of the object is replicated across replicas. Clients can call methods at every replica and the calls are communicated between replicas. The replicated object is expected to satisfy both consistency and convergence. *Consistency* is the safety property that every method call is executed only when the guard of the method and the invariant are satisfied. *Convergence* is the safety property that when no call is in transit, all replicas converge to the same state. We want to perform coordination only when necessary to preserve these properties. We say that a method call c is *permissible* in a state σ, written as $\mathcal{P}(\sigma, c)$, if the guard of c is satisfied in σ and c results in a post-state σ' that satisfies the invariant $\mathcal{I}$ that is $\mathcal{I}(\sigma')$. The post-state of a method call is the pre-state of the next in a replica. The initial state is assumed to satisfy the invariant. Therefore, if every call is permissible in its pre-state, then every call is consistent. To execute a method call, we check that it is permissible in its originating replica. Thus, we say that each method call is *locally permissible*. Otherwise, the call is aborted. Still, if the call is simply broadcast, it is not necessarily permissible when it arrives at other replicas. Some calls need coordination. We now present representative incorrect executions to showcase the conditions that necessitate coordination.

Well-coordination. Method calls such as adding a course and enrolling a student result in the same state if their order of execution is swapped. However, the resulting state of some pairs of methods calls is dependent on their execution order. Fig. 2.(a) shows an execution where a course c is added and deleted concurrently at two replicas. The two method calls are executed without coordination and are broadcast to other replicas and executed on arrival. Thus, the two replicas execute the two method calls in two different orders and their final states diverge. Reordering the execution of adding and removing a value from a set does not result in the same state. (As we will see in § 5, particular CRDT sets can converge even when their operations reorder [Shapiro et al. 2011].) As Fig. 2.(b) shows, we say that two method calls $\mathcal{S}$*-commute* (state-commute) written as $c_1 \leftrightarrows_{\mathcal{S}} c_2$, iff starting from the same pre-state, executing them in either of the two orders results in the same post-state. Otherwise, we say that they $\mathcal{S}$*-conflict* (state-conflict) and need synchronization; they should be executed one at a time so that they have the same order across replicas.

A method call such as registering a student always preserves the invariant. It adds a student and cannot result in a missing student or course in the enrolment relation. Thus, if it is broadcast and executed on a replica whose state satisfies the invariant, it preserves the invariant. We call such method calls *invariant-sufficient*. However, not all method calls are invariant-sufficient. Fig. 2.(c) shows an execution where the enrolment of a student s in a course c is executed in the first replica. This method call preserves the invariant as both the student s and the course c belong to the corresponding relations. A method call that deletes the course c is executed concurrently in the

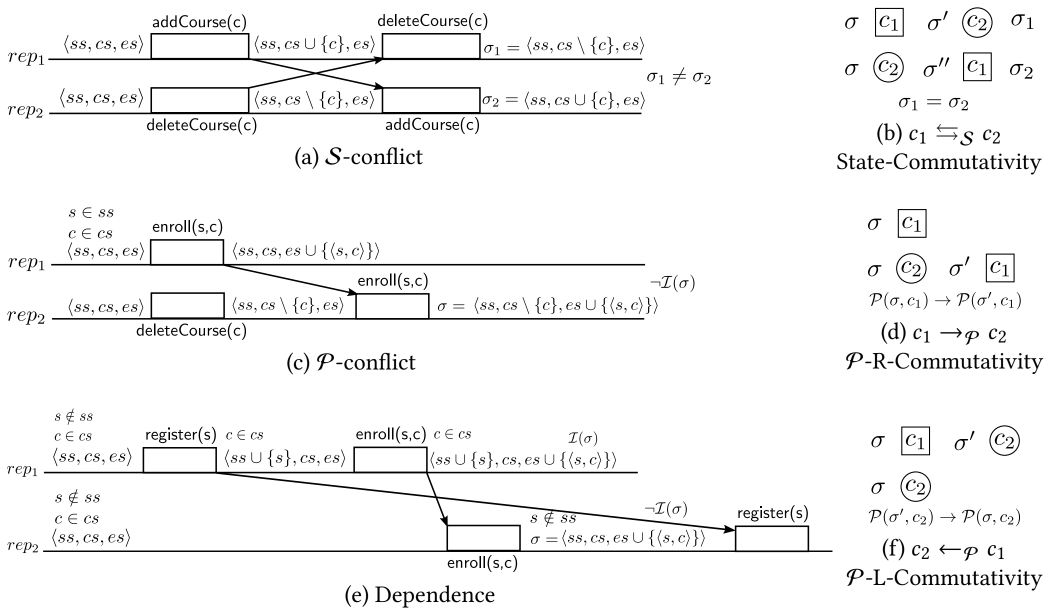

addCourse(c)
deleteCourse(c)
rep_1
rep_2
$\langle ss, cs, es\rangle$
$\langle ss, cs \cup \{c\}, es\rangle$
$\sigma_1 = \langle ss, cs \setminus \{c\}, es\rangle$
$\langle ss, cs \setminus \{c\}, es\rangle$
$\sigma_2 = \langle ss, cs \cup \{c\}, es\rangle$
$\sigma_1 \neq \sigma_2$
(a) $\mathcal{S}$-conflict
σ $\boxed{c_1}$ σ' c_2 σ_1
σ c_2 σ'' $\boxed{c_1}$ σ_2
$\sigma_1 = \sigma_2$
(b) $c_1 \leftrightarrows_{\mathcal{S}} c_2$
State-Commutativity
$s \in ss$
$c \in cs$
enroll(s,c)
$\langle ss, cs, es \cup \{\langle s, c\rangle\}\rangle$
$\neg\mathcal{I}(\sigma)$
$\sigma = \langle ss, cs \setminus \{c\}, es \cup \{\langle s, c\rangle\}\rangle$
(c) $\mathcal{P}$-conflict
σ $\boxed{c_1}$
σ c_2 σ' $\boxed{c_1}$
$\mathcal{P}(\sigma, c_1) \rightarrow \mathcal{P}(\sigma', c_1)$
(d) $c_1 \rightarrow_{\mathcal{P}} c_2$
$\mathcal{P}$-R-Commutativity
$s \notin ss$
register(s)
$\langle ss \cup \{s\}, cs, es\rangle$
$\mathcal{I}(\sigma)$
$\langle ss \cup \{s\}, cs, es \cup \{\langle s, c\rangle\}\rangle$
$\sigma = \langle ss, cs, es \cup \{\langle s, c\rangle\}\rangle$
(e) Dependence
σ $\boxed{c_1}$ σ' c_2
σ c_2
$\mathcal{P}(\sigma', c_2) \rightarrow \mathcal{P}(\sigma, c_2)$
(f) $c_2 \leftarrow_{\mathcal{P}} c_1$
$\mathcal{P}$-L-Commutativity

Fig. 2. Incorrect Executions and Coordination Avoidance Conditions. Square and circle around method calls in (b) , (d) and (f) are just visual aids to see the movements.

second replica. The enroll call is broadcast and received at the second replica after the delete call. It does not preserve the invariant at the second replica as it is enrolling in a missing course. These two method calls should synchronize to preserve the invariant. Nonetheless, some pairs of method calls such as enrolling in a course and adding the course do not need synchronization. We say that the call c_1 *$\mathcal{P}$-R-commutes* (permissible-right-commutes) with the call c_2 written as $c_1 \rightarrow_{\mathcal{P}} c_2$, iff c_1 stays permissible if it is moved right after c_2. More precisely, as Fig. 2.(d) shows, for every state σ, if c_1 is permissible in σ, then it is permissible after applying c_2 to σ as well. We say that a method call c_1 *$\mathcal{P}$-concurs* (permissible-concurs) with another call c_2 iff either c_1 is invariant-sufficient or c_1 $\mathcal{P}$-R-commutes with c_2. Otherwise, we say that c_1 *$\mathcal{P}$-conflicts* (permissible-conflict) with c_2 and they need synchronization. Enrolling in a course $\mathcal{P}$-concurs with adding the course; however, enrolling in a course $\mathcal{P}$-conflicts with deleting the course, therefore; they should synchronize.

We say that two method calls *concur* iff they both $\mathcal{S}$-commute and $\mathcal{P}$-concur with each other. Otherwise, we say they *conflict* and need synchronization. We statically analyze methods of the object and determine whether they satisfy these properties. Fig. 1.(b) and (c) show the result of the analysis for $\mathcal{S}$-commute and $\mathcal{P}$-concur on the courseware use-case and based on them, Fig. 1.(d) shows the concur relation. The conflict relation is the complement of the concur relation. Fig. 1.(e) shows the conflict graph where edges connect pairs of conflicting methods. In our running example, deleting a course conflicts with adding a course and enrolment.

As explained above, invariant-sufficient method calls always preserve the invariant. However, there are calls whose preservation of the invariant is dependent on the calls that have executed before them at that replica. Fig. 2.(e) shows an execution where a student is registered and subsequently enrolled in a course. The method calls are broadcast, reordered during transmission and executed in the opposite order in the second replica. The invariant holds after the enrolment in the first replica as it enrolls an existing student in a course. The student has been just registered. However, the enrolment violates the invariant in the second replica. As the student is enrolled before she is registered, a missing student is enrolled which violates the referential integrity of the enrolment relation. Nonetheless, an enrolment is independent of other enrolments. We say that a method call c_2 *$\mathcal{P}$-L-commutes* (permissible-left-commutes) with a method call c_1 written as $c_2 \leftarrow_{\mathcal{P}} c_1$, iff c_2 remains permissible if it is moved left before c_1. More precisely, as Fig. 2.(f) shows, for every state σ, if c_2 is permissible in the state resulted from executing c_1 on σ, then c_2 is permissible in σ as well. We say that a method call c_2 is *independent* of c_1 iff c_2 is either invariant-sufficient or $\mathcal{P}$-L-commutes with c_1. The dependencies of a method call is the set of method calls that it is dependent on. If c_2 is dependent on c_1 and c_1 is executed before c_2 in the originating replica of c_2, then c_2 should be applied to other replicas only if c_1 is already applied. In Fig. 2.(e), the enrolment is not invariant-sufficient and does not $\mathcal{P}$-L-commute with the registration of the student; thus, the enrolment is dependent on the registration. The enrolment in the second replica should be postponed to after the student is registered. Nonetheless, an enrolment $\mathcal{P}$-L-commutes with other enrolments. Fig. 1.(f) shows the result of static analysis for the independence relation on the courseware use-case. The dependence relation is the complement of the independence relation. The dependence graph is shown in Fig. 1.(g). Enrolment is dependent on registration and adding a course.

We say that an execution is *conflict-synchronizing* if the same order is enforced for conflicting method calls across all replicas. We say that an execution is *dependency-preserving* if every method call is executed only after its dependencies from its originating replica are already executed. We define *well-coordinated* executions as locally permissible, conflict-synchronizing and dependency-preserving executions. In § 3, we formally define well-coordination and prove that it is sufficient for consistency and convergence of replicas.

Protocols. We now outline our protocols that provide well-coordination and are used to synthesize replicated objects. For the given object, a static analysis finds the conflict and dependency relations that we saw above. The analysis results are used to instantiate the protocols. In this overview, we assume that methods are independent and focus on synchronization of conflicting methods. We outline two protocols. The first protocol is non-blocking and makes progress even if some replicas crash. The second protocol is blocking but can execute calls on one method of a conflicting pair without synchronization.

Non-Blocking Protocol. The high-level idea is to find sets of conflicting calls and synchronize calls in each set. We remember that a clique is a subset of the vertices of a graph such that any of its distinct pair of vertices are adjacent. We find the maximal cliques of the conflict graph and synchronize the methods of each maximal clique with each other. For example, in the conflict graph of the courseware use-case shown in Fig. 1.(e), the maximal cliques are $cl_1 = \{d, a\}$ and $cl_2 = \{d, e\}$ where d is deletion, a is addition and e is enrolment. Deletion d and addition a and also deletion d and enrolment e should synchronize with each other. Deletion d is a member of two cliques and should synchronize in both. We use a variant of the classical total-order broadcast (TOB) protocol to deliver method calls in the same order at all replicas. We use a TOB instance for each maximal clique. In our example, we use the TOB instances tob_1 and tob_2 for the cliques cl_1 and cl_2. A call on a method such as d that is a member of multiple maximal cliques should be totally ordered with respect to methods of each of those cliques. The call is broadcast to each TOB instance and is executed only when it is ordered and delivered by all of them. The non-blocking property of the protocol is derived from the termination property of TOB when a majority of nodes are correct.

As an example, Fig. 3.(a) shows an execution of the protocol on the courseware use-case. Three methods are called at three replicas: adding a a course c, enrolling e a student s in the course c and deleting d the course c. The call a is broadcast using tob_1, and the call e is broadcast using tob_2. The call d has to be broadcast to both tob_1 and tob_2. It is first broadcast to tob_1. The sub-protocol tob_1 decides to order and deliver a before d. Thus, a is delivered first and executed at the three replicas.

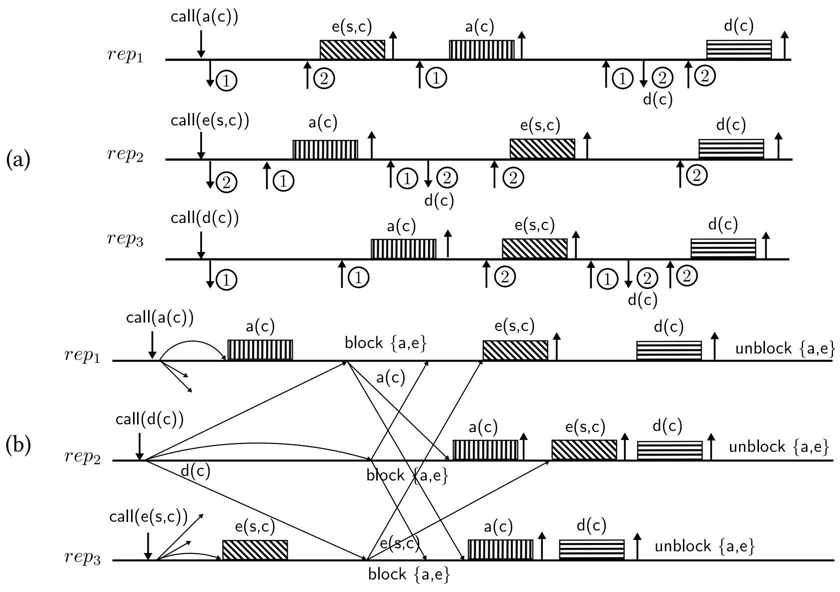


Fig. 3. (a) Non-blocking Synchronization Protocol. The symbols ↓ and ↑ show requests to and responses from the protocols. Events to the main protocol are shown above and events to the sub-protocols are shown below the horizontal time line. The symbols ① and ② represent events of the first and second TOB sub-protocols respectively. Blocks show the execution of method calls. (b) Blocking Synchronization Protocol. The symbols ↓ and ↑ show requests to and responses from the protocols. Diagonal arrows show message transmission.

The sub-protocol tob_2 independently delivers e. It is notable that the execution order of e and a that belong to distinct cliques and are broadcast to distinct TOB instances are different in the first and the second replica. Once d is delivered by tob_1, it is broadcast to tob_2. It is finally delivered by tob_2 as well and executed. Thus, the call d is finally executed after both a and e at all replicas.

In the above execution, when the call d is delivered by tob_1, it is implicitly assigned a particular place in the total order of calls in the first clique. However, it cannot execute on delivery from tob_1 and should be broadcast by tob_2. To keep the place of d, other calls delivered by tob_1 should wait for d to finish its synchronization in the second clique. Therefore, we use a queue per TOB. Method calls that are delivered by a TOB are enqueued to its corresponding queue. A call should wait and can be executed only when it appears at the head of the queues of all TOBs that it is broadcast to. Unfortunately naive implementation of waiting can potentially make mutual waiting and deadlocks. For example, two calls on d can be ordered differently by tob_1 and tob_2 and wait for each other in a deadlock. In § 6.1, we revisit this problem and present and use a novel variant of TOB called multi-total-order broadcast (MTOB) that prevents deadlocks.

Blocking Protocol. If two method calls conflict, the previous protocol requires both to go through synchronization. We now present an overview of a protocol that pushes synchronization to only one of the two. Consider that there are two conflicting methods m and m' and we want to let calls on m execute without synchronization. The idea is that calls on m' reach out to other replicas, block the execution of calls on m (so that new calls on m are not accepted) and then replicas exchange updates on preceding calls on m. Once the replicas apply the updates, they have the same set of executed calls on m. Then, the call on m' is executed at all replicas and calls on m are unblocked.

We remember that a minimum vertex cover of a graph is a smallest subset of the vertices such that every edge has at least one endpoint in the cover. To avoid synchronization, we find a minimum vertex cover of the conflict graph and synchronize only when methods in the cover are called. For example, in the conflict graph of the courseware use-case, shown in Fig. 1.(e), the minimum vertex cover is the singleton set of the delete method $\{d\}$. Only deletion d performs synchronization and addition a and enrolment e can execute without synchronization. Further, methods can be assigned weights inversely proportional to their call frequency and weighted minimum vertex cover can optimize the average responsiveness of the replicated object.

As an example, Fig. 3.(b) shows an execution of the protocol on the courseware use-case. The first and the third replicas call synchronization-free methods a and e. They are simply broadcast and executed on arrival. In this execution, the delivery of these messages are delayed. The second replica calls method d. The call d is broadcast and on its delivery, all replicas block the conflicting methods a and e. To update other replicas, each replica subsequently broadcasts the set of conflicting method calls that it has executed. The first and third replicas broadcast their calls on a and e respectively. These updates are applied on arrival. After all the updates are applied, every replica has executed the same set of calls that conflict with d although possibly in different orders. Then, the call on d is executed and the conflicting methods are unblocked. This protocol makes replicas wait for each other; thus, crash of a replica can prevent progress of others. Following fundamental impossibility results [Fischer et al. 1985; Gilbert and Lynch 2002], this protocol has a trade-off between availability and consistency. We will revisit this trade-off in § 6.2.

3 WELL-COORDINATION

In this section, we define the well-coordination condition and prove that it is sufficient for state integrity and convergence. We first define replicated executions and their correctness. Then, we present the well-coordination conditions and prove that well-coordinated executions are correct.

An object is a record $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ that includes a state type Σ, an invariant $\mathcal{I}$ on the state, and a set of methods $\mathcal{M}$. A method is a function m from the parameters and the pre-state to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$, where guard is a boolean expression that defines when the method can be called, and update and retv are expressions for the post-state and the return value. We use guard, update and retv as functions that extract elements of the record. A method call c is a method applied to its argument i.e. a function from the current state to a record of $\langle \mathsf{guard}, \mathsf{update}, \mathsf{retv} \rangle$.

Execution. We first define the context $\mathbf{c}$ for a replicated execution. The state of each replica is initialized to the same state σ_0 that satisfies the invariant $\mathcal{I}$. The user can request a call on a method at every replica that is called the originating replica of the call. The call is then propagated from the originating replica and executed at other replicas. We uniquely identify requests by identifiers.

Definition 1 (Execution Context). *An execution context* $\mathbf{c}$ *is the record* $\langle \sigma_{0\mathbf{c}}, R_{\mathbf{c}}, \mathit{call}_{\mathbf{c}}, \mathit{orig}_{\mathbf{c}} \rangle$ *where* $\sigma_{0\mathbf{c}}$ *is an initial state that satisfies the invariant i.e.* $\mathcal{I}(\sigma_{0\mathbf{c}})$*,* $R_{\mathbf{c}}$ *is a set of request identifiers,* $\mathit{call}_{\mathbf{c}}$ *is a function from* $R_{\mathbf{c}}$ *to method calls, and* $\mathit{orig}_{\mathbf{c}}$ *is a function from* $R_{\mathbf{c}}$ *to replicas* $\mathcal{N}$*.*

We model an execution at a replica as a permutation of a set of request identifiers.

Definition 2 (Execution). *In a context* $\mathbf{c}$*, an execution* x *of a set of requests* $R \subseteq R_{\mathbf{c}}$ *is a bijective from positions* $[0..|R| - 1]$ *to* R*.*

We denote the range of x *as* $R(x)$*. An execution* x *of* R *defines the total order* $\prec_x$ *on* R*. A request* r *precedes another request* r' *in an execution* x *written as* $r \prec_x r'$ *iff* $x^{-1}(r) < x^{-1}(r')$*.*

In a replicated execution, calls are propagated and eventually executed at every replica. Convergence is a condition on the state of the replicas after all calls are applied at all replicas. Therefore, a replicated execution is a mapping from replicas to permutations of the same set of calls. For example, Fig. 4.(a) shows a replicated execution where nine requests are executed. Propagation of calls from the originating replicas to other replicas creates a visibility relation between calls across replicas. For example, in Fig. 4.(a), arrows show the visibility relation. Consequently, the happens-before relation is the transitive closure of the visibility relation and the execution order of each replica. The happens-before relation is acyclic. In Fig. 4.(a), as the direction of all arrows is forward, the happens-before relation is acyclic.

Definition 3 (Replicated Execution). *In a context* $\mathbf{c}$*, a replicated execution* xs *is a function from replicas* $\mathcal{N}$ *to executions of* $R_{\mathbf{c}}$ *such that (1) let the execution order* $\prec_{xs}$ *on* $\mathcal{N} \times R_{\mathbf{c}}$ *be defined as: for every replica* n *and pair of requests* r *and* r'*,* $(n, r) \prec_{xs} (n, r')$ *iff* $r \prec_{xs(n)} r'$*, (2) let the visibility relation* $\rightsquigarrow_{xs}$ *on* $\mathcal{N} \times R_{\mathbf{c}}$ *be defined as: for every request* r*, for every replica* n*,* $(orig_{\mathbf{c}}(r), r) \rightsquigarrow_{xs} (n, r)$ *iff* $n \neq orig_{\mathbf{c}}(r)$*, (3) let the happens-before relation* hb_{xs} *be* $(\prec_{xs} \cup \rightsquigarrow_{xs})^*$ *then,* hb_{xs} *is acyclic.*

The post-state of each call at a replica is the result of applying the call to its pre-state. Thus, a sequence of calls result in a sequence of states.

Definition 4 (State). *In a context* $\mathbf{c}$*, the state function* $\mathbf{s}$ *of an execution* x *is a function from positions* $[0..|R(x)|]$ *to states* Σ *such that* $\mathbf{s}(0) = \sigma_{0\mathbf{c}}$ *and for every* $0 \leq i < |R(x)|$*,* $\mathbf{s}(i+1) = update(call_{\mathbf{c}}(x(i)))(\mathbf{s}(i))$*. The state function is lifted to replicated executions. The state function* $\mathbf{ss}$ *of a replicated execution* xs *is a function from replicas* n *in* $\mathcal{N}$ *to the state function of the execution* $xs(n)$*.*

Correctness. We now define correctness as convergence and integrity.
A replicated execution is convergent if it leads to the same final state for all replicas.

Definition 5 (Convergent). *A replicated execution* xs *of a context* $\mathbf{c}$ *is convergent iff for every pair of replicas* n *and* n'*,* $\mathbf{ss}(n)(|R_{\mathbf{c}}|) = \mathbf{ss}(n')(|R_{\mathbf{c}}|)$ *where* $\mathbf{ss}$ *is the state function of* xs*.*

In the definition of methods of an object, the user relies on the invariant in the pre-state. Further, methods have explicit guards that define the subset of states that they are applicable to. We say that a method call is consistent at a state if the invariant and the guard of the method hold in that state. Method calls should be executed only on states that they are consistent in.

Definition 6 (Consistent Call). *A method call* c *is consistent in a state* σ*, written as* $cons(\sigma, c)$*, iff* $guard(c)(\sigma)$ *and* $\mathcal{I}(\sigma)$*.*

The consistency condition is simply lifted to executions and replicated executions.

Definition 7 (Consistent Execution). *In a context* $\mathbf{c}$*, a request* r *is consistent in an execution* x *written as* $cons(\mathbf{c}, x, r)$ *iff* $cons(\mathbf{s}(i), call_{\mathbf{c}}(r))$ *where* $\mathbf{s}$ *is the state function of* x*, and* i *is* $x^{-1}(r)$*. In a context* $\mathbf{c}$*, an execution* x *is consistent written as* $cons(\mathbf{c}, x)$ *iff every request* r *in* $R(x)$ *is consistent in* x*. A replicated execution* xs *of a context* $\mathbf{c}$ *is consistent written as* $cons(\mathbf{c}, xs)$ *iff for every replica* n*, the execution* $xs(n)$ *is consistent.*

Consistency of a replicated execution requires invariant preservation (that is state integrity) at all replicas. We define correctness as both consistency and convergence.

Definition 8 (Correct). *A replicated execution is correct iff it is consistent and convergent.*

Well-coordination. Now, we define the well-coordination conditions. We say that a call is permissible in a state iff its guard holds in that state and the invariant holds after the call is applied.

Definition 9 (Permissible Call). *A method call* c *is permissible in a state* σ*, written as* $\mathcal{P}(\sigma, c)$*, iff* $guard(c)(\sigma)$ *and* $\mathcal{I}(update(c)(\sigma))$*.*

Note that in contrast to the definition of consistency above that requires the invariant to hold in the pre-state, permissibility requires it to hold in the post-state. By induction, permissibility leads to consistency. The initial state satisfies the invariant; thus, for every call, if all the previous calls have maintained the invariant, the call is applied to a state that satisfies the invariant as well. Permissibility implies that the call preserves the invariant. Similar to consistency, permissibility is simply lifted to executions and replicated executions. For brevity, we elide this to the appendix § 1 [Appendix 2018].

Well-coordination requires each call to be permissible in its originating replica. If a call is requested at a replica but is not permissible in its current state, the call should be aborted (and maybe retried later).

Definition 10 (Locally permissible). *A replicated execution xs of a context* $\mathbf{c}$ *is locally permissible iff every request r is permissible in the execution of its originating replica $orig_{\mathbf{c}}(r)$.*

Although permissibility is directly checked only locally at the originating replicas, we will show that well-coordination conditions ensure the global permissibility of calls at every replica.

As we saw in Fig. 2.(b), we say that two method calls $\mathcal{S}$-commute (state-commute) if starting from every pre-state, the post-state is the same if the calls are reordered.

Definition 11 (State-Commutativity and State-Conflict). *Two method calls c_1 and c_2 S-commute, written as $c_1 \leftrightarrows_{\mathcal{S}} c_2$ iff for every state σ, $update(c_2)(update(c_1)(\sigma)) = update(c_1)(update(c_2)(\sigma))$. Otherwise, they $\mathcal{S}$-conflict, written as $c_1 \bowtie_{\mathcal{S}} c_2$.*

$\mathcal{S}$-conflicting calls need synchronization since we saw in Fig. 2.(a) that they cause state divergence.

We note that $\mathcal{S}$-commutativity and the following properties are defined on (dynamic) method calls; however, they are simply lifted to (static) methods. For instance, we say that two methods $\mathcal{S}$-commute iff all calls on the two $\mathcal{S}$-commute. In § 4, we consider these properties on methods.

There are calls such as deposit on a bank account that are always permissible as far as they are applied to a state that satisfies the invariant. We call these calls invariant-sufficient.

Definition 12 (Invariant-Sufficient). *A call c is invariant-sufficient iff for every state σ if $\mathcal{I}(\sigma)$ then $\mathcal{P}(\sigma, c)$.*

Every call is checked to be permissible in its originating replica. However, as we saw in Fig. 2.(c), if a call is simply broadcast, when it arrives at other replicas, other calls may have been executed at the destination replicas that were not executed at the originating replica. These extra calls may make the arrived call impermissible. As we saw in Fig. 2.(d), we say that a method call $\mathcal{P}$-R-commutes (permissible-right-commutes) another if starting from any state where the former is permissible, moving it right after the latter does not violate permissibility.

Definition 13 (Permissible-Right-Commutativity). *The call c_1 $\mathcal{P}$-R-commutes with the call c_2 written as $c_1 \rightarrow_{\mathcal{P}} c_2$ iff for every state σ, if $\mathcal{P}(\sigma, c_1)$ then $\mathcal{P}(update(c_2)(\sigma), c_1)$.*

If a call is invariant-sufficient or $\mathcal{P}$-R-commutes another call, we say that the former $\mathcal{P}$-concurs (permissible-concurs) with the latter. Otherwise, we say that the former $\mathcal{P}$-conflicts (permissible-conflicts) with the latter.

Definition 14 (Permissible-Concur and Permissible-Conflict). *A call c_1 $\mathcal{P}$-concurs with a call c_2 iff c_1 is invariant-sufficient or $c_1 \rightarrow_{\mathcal{P}} c_2$. Otherwise, c_1 $\mathcal{P}$-conflicts with c_2.*

A pair of calls can avoid synchronization only if they both $\mathcal{S}$-commute and $\mathcal{P}$-concur with respect to each other.

Definition 15 (Concur and Conflict). *A pair of calls c_1 and c_2 concur iff they $\mathcal{S}$-commute and $\mathcal{P}$-concur with each other. Otherwise, they conflict written as $c_1 \bowtie c_2$.*

Concur and conflict relations are symmetric. The conflict relation on methods can be represented as the conflict graph $G_{\bowtie}$: an undirected graph where the vertices are the set of methods and the edges are the pairs of conflicting methods. A replicated execution is conflict-synchronizing if every pair of conflicting calls have the same order across replicas.

Definition 16 (Conflict-synchronizing). *A replicated execution xs of a context* $\mathbf{c}$ *is conflict-synchronizing iff for every pair of requests* r *and* r' *in* $R_{\mathbf{c}}$ *such that* $call_{\mathbf{c}}(r) \bowtie call_{\mathbf{c}}(r')$, *for every pair of replicas* n *and* n', *if* $r \prec_{xs(n)} r'$ *then* $r \prec_{xs(n')} r'$.

Similar to conflict-synchronizing, $\mathcal{S}$-conflict-synchronizing and $\mathcal{P}$-conflict-synchronizing are similarly defined with respect to $\mathcal{S}$-conflict and $\mathcal{P}$-conflict. (We elide them to the appendix).

As we saw in Fig. 2.(e), when a call arrives at other replicas, other calls that were executed at the originating replica may have not arrived and executed at destination replicas. However, permissibility of the call may be dependent on the missing calls. As we saw in Fig. 2.(f), we say that a method call $\mathcal{P}$-L-commutes (permissible-left-commutes) with another if moving the former left before the latter does not render the former impermissible.

Definition 17 (Permissible-Left-Commutative). *A call* c_2 $\mathcal{P}$*-L-commutes a call* c_1, *written as* $c_2 \leftarrow_{\mathcal{P}} c_1$ *iff for every state* σ, *if* $\mathcal{P}(update(c_1)(\sigma), c_2)$ *then* $\mathcal{P}(\sigma, c_2)$.

A call can avoid tracking dependencies to another call if the former is invariant-sufficient or $\mathcal{P}$-L-commutes with the latter.

Definition 18 (Independent and Dependent). *A call* c_2 *is independent of* c_1, *written as* $c_2 \perp\!\!\!\perp c_1$, *iff either* c_2 *is invariant-sufficient or* $c_2 \leftarrow_{\mathcal{P}} c_1$. *Otherwise,* c_2 *is dependent on* c_1, *written as* $c_2 \not\perp\!\!\!\perp c_1$.

The dependency relation between methods can be represented as a directed graph that we call the dependency graph. A replicated execution is dependency-preserving if for every call, its preceding dependencies in its originating replica precede it in the other replicas as well.

Definition 19 (Dependency-Preserving). *A replicated execution xs of a context* $\mathbf{c}$ *is dependency-preserving iff for every pair of requests* r *and* r' *in* $R_{\mathbf{c}}$, *such that* $call_{\mathbf{c}}(r') \not\perp\!\!\!\perp call_{\mathbf{c}}(r)$, *if* $r \prec_{xs(orig_{\mathbf{c}}(r'))} r'$, *then for every replica* n, $r \prec_{xs(n)} r'$.

We note that in Def. 16, call orders in any replica necessitates the same orders in other replicas. In contrast, in Def. 19, only orders between a call and its preceding calls in its *originating replica* necessitates the same order in other replicas.

A replicated execution is well-coordinated if the permissibility of calls are checked at the originating replicas, conflicting calls are synchronized and the dependencies are preserved. Well-coordination is a sufficient condition for the correctness of replicated executions.

Definition 20 (Well-coordination). *A replicated execution is well-coordinated iff it is locally permissible, conflict-synchronizing, and dependency-preserving.*

Theorem 1. *Every well-coordinated replicated execution is correct.*

The full proof is available in the appendix § 1. It follows from the definition of well-coordination and correct (Def. 20 and Def. 8) and the following two lemmas. We present the high-level ideas.

Lemma 1. *Every* $\mathcal{S}$*-conflict-synchronizing replicated execution is convergent.*

Consider two executions x and x′ from the replicated execution (with the same set of requests possibly in different orders). Assume that x and x′ are $\mathcal{S}$-conflict-synchronizing with respect to each other. We prove that these two executions result in the same post-state. By induction, x′ can be incrementally converted to x from left to right without changing its final post-state. Assume

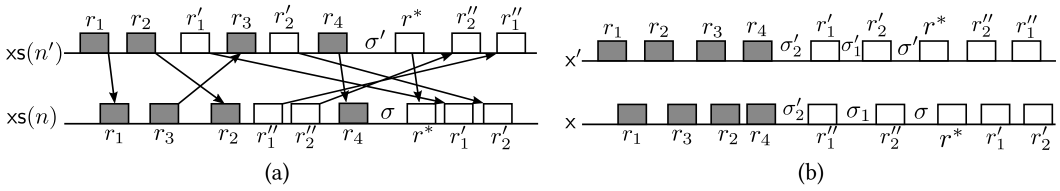

r_1 r_2 r'_1 r_3 r'_2 r_4 σ' r^* r''_2 r''_1
xs(n')
xs(n)
r_1 r_3 r_2 r''_1 r''_2 r_4 σ r^* r'_1 r'_2
(a)
r_1 r_2 r_3 r_4 σ'_2 r'_1 σ'_1 r'_2 σ' r^* r''_2 r''_1
x′
x
σ'_2 σ_1 σ
r_1 r_3 r_2 r_4 r''_1 r''_2 r^* r'_1 r'_2
(b)

Fig. 4. Correctness of well-coordinated replicated executions

that the requests until location i are the same in x and x′. Consider the request r at position i in x. If r appears later at position j in x′ where $j > i$, then we show that r can be moved left in x′ to position i. The requests between i and j in n' precede r in x′ but succeed r in x. Therefore, by the $\mathcal{S}$-conflict-synchronization condition, r $\mathcal{S}$-commutes with requests between i and j in x′. Thus, r can be moved left to location i in x′ without any change to the post-state. ■

Lemma 2. *Every well-coordinated replicated execution is consistent.*

We illustrate the crucial part of the proof by a figure. Let xs be a coordinated replicated execution. To prove consistency of xs, we need to prove consistency of every request at the execution of every replica. We will prove that every request at every replica is permissible. This implies that (1) the guard of every request is satisfied. and (2) the post-state of every request satisfies the invariant. Based on [2] and the fact that the initial state is defined to satisfy the invariant, we have that (3) the pre-state of every request satisfies the invariant. From the facts [1] and [3] above, we have that xs is consistent. We now show the permissibility of every request r^*. The proof is by induction on a linear extension of $\mathsf{hb}_{\mathsf{xs}}$. Let the request r^* at the replica n be the current request. If n is the originating replica of r^*, then r^* is trivially permissible by the locally permissible condition; it states that every replica only originates permissible requests. Otherwise, let n' be the originating replica of r^*. If r^* is invariant-sufficient, we only need to show that the pre-state of r^* in n satisfies the invariant. The pre-state of r^* is either the initial state that by definition satisfies the invariant or is the post-state of the preceding request in n. By the induction hypothesis, the preceding request is permissible that implies that its post-state satisfies the invariant.

Now we consider that r^* is not invariant-sufficient. We illustrate the proof of permissibility of r^* in Fig. 4. Let σ be the pre-state of r^* in xs(n). We want to show that r^* is permissible in σ. Let σ' be the pre-state of r^* in xs(n') (the execution of the originating replica). Let R be the requests that precede r^* in both xs(n) and xs(n'). In Fig. 4.(a), R is the set of shaded requests $\{r_1, r_2, r_3, r_4\}$. Let R' be the requests that precede r^* in $xs(n')$ but *do not* precede r^* in xs(n). In Fig. 4.(a), R' is $\{r'_1, r'_2\}$. Consider a request r in R and a request r' in R' such that r' precedes r in xs(n'). In Fig. 4.(a), r can be r_4 and r' can be r'_2. The request r' precedes r in xs(n') but succeeds it in xs(n). Therefore, by the $\mathcal{S}$-conflict-synchronization condition, r' and r $\mathcal{S}$-commute. In Fig. 4.(a), we commute r'_2 with r_4. Then, we commute r'_1 with r_3 and r_4. Thus, by induction, each request in R' from the rightmost to the leftmost in xs(n') can be moved right to form a block of requests before r^* in xs(n') without changing the pre-state σ' of r^*. Let x′ denote the result of the commute. Fig. 4.(b) shows x′ where the pre-state of r^* is still σ'. In Fig. 4.(a), the requests R' precede r^* in xs(n') but succeed it in xs(n). Therefore, by the dependency-preserving condition, r^* is independent of the requests in R'. In Fig. 4.(a), r^* is independent of r'_1 and r'_2. By the locally permissible condition and that n' is the originating replica of r^*, the request r^* is permissible at its pre-state σ' in xs(n'). By induction from right to left in x′, using the independence condition, r^* is permissible at the pre-state of each request r' in R'. Thus, r^* is permissible at the pre-state of R' that is the post-state of R in x′. In Fig. 4.(b), r^* is permissible at the states σ'_1 and σ'_2.

```
    fun ConflictRel(): $\mathcal{M} \times \mathcal{M} \to \mathbb{B}$ {
$C_1$      var SCom: $\mathcal{M} \times \mathcal{M} \to \mathbb{B}$
$C_2$      var ISuff: $\mathcal{M} \to \mathbb{B}$
$C_3$      var PRCom, PConcur: $\mathcal{M} \times \mathcal{M} \to \mathbb{B}$
$C_4$      var Concur, Conflict: $\mathcal{M} \times \mathcal{M} \to \mathbb{B}$
$C_5$      let $\mathcal{P} := \lambda\sigma, c.\ \text{guard}(c)(\sigma) \wedge \mathcal{I}(\text{update}(c)(\sigma))$
$C_6$      foreach $(m_1 \in \mathcal{M}, m_2 \in \mathcal{M})$
$C_7$          SCom$(m_1, m_2)$ :=
                   $\vdash \forall\sigma, a_1, a_2$
                       update$(m_2(a_2))$(update$(m_1(a_1))(\sigma)) =$
                       update$(m_1(a_1))$(update$(m_2(a_2))(\sigma))$
$C_8$      foreach $(m \in \mathcal{M})$
$C_9$          ISuff$(m)$ :=
                   $\vdash \forall\sigma, a.\ \mathcal{I}(\sigma) \to \mathcal{P}(\sigma, m(a))$
$C_{10}$   foreach $(m_1 \in \mathcal{M}, m_2 \in \mathcal{M})$
$C_{11}$       PRCom$(m_1, m_2)$ :=
                   $\vdash \forall\sigma, a_1, a_2.$
                       $\mathcal{P}(\sigma, m_1(a_1)) \to$
                       $\mathcal{P}(\text{update}(m_2(a_2))(\sigma), m_1(a_1))$
$C_{12}$       PConcur$(m_1, m_2)$ := ISuff$(m_1)$ or
                                       PRCom$(m_1, m_2)$
$C_{13}$   foreach $(m_1 \in \mathcal{M}, m_2 \in \mathcal{M})$
$C_{14}$       Concur$(m_1, m_2)$ := SCom$(m_1, m_2)$ and
                                     PConcur$(m_1, m_2)$ and
                                     PConcur$(m_2, m_1)$
$C_{15}$       Conflict$(m_1, m_2)$ := not Concur$(m_1, m_2)$
$C_{16}$   return Conflict }

    fun DepRel(): $\mathcal{M} \times \mathcal{M} \to \mathbb{B}$ {
$D_1$      var ISuff: $\mathcal{M} \to \mathbb{B}$
$D_2$      var LRCom: $\mathcal{M} \times \mathcal{M} \to \mathbb{B}$
$D_3$      var Dep, Indep: $\mathcal{M} \times \mathcal{M} \to \mathbb{B}$
$D_4$      let $\mathcal{P} := \lambda\sigma, c.\ \text{guard}(c)(\sigma) \wedge \mathcal{I}(\text{update}(c)(\sigma))$
$D_5$      foreach $(m \in \mathcal{M})$
$D_6$          ISuff$(m)$ :=
                   $\vdash \forall\sigma, a.\ \mathcal{I}(\sigma) \to \mathcal{P}(\sigma, m(a))$
$D_7$      foreach $(m_2 \in \mathcal{M}, m_1 \in \mathcal{M})$
$D_8$          PLCom$(m_2, m_1)$ :=
                   $\vdash \forall\sigma, a_1, a_2.$
                       $\mathcal{P}(\text{update}(m_1(a_1))(\sigma), m_2(a_2)) \to$
                       $\mathcal{P}(\sigma, m_2(a_2))$
$D_9$          Indep$(m_2, m_1)$ := ISuff$(m_2)$ or
$D_{10}$                              PLCom$(m_2, m_1)$
$D_{11}$       Dep$(m_2, m_1)$ := not Indep$(m_2, m_1)$
$D_{12}$   return Dep }
```

Fig. 5. Static analysis to calculate the conflict and dependency relations. The object $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ is given.

The argument above for moving requests in xs(n') can be applied to xs(n) as well. Let R'' be the requests that precede r^* in $xs(n)$ but do not precede it in $xs(n')$. In Fig. 4.(a), R'' is $\{r''_1, r''_2\}$. $\mathcal{S}$-commutativity allows moving R'' right in xs(n). The requests R'' can be moved to form a block immediately before r^* without changing the pre-state of r^*. Let x denote the result of the commute. Fig. 4.(b) shows x. The requests $\{r''_1, r''_2\}$ moved right immediately before r^*. The set of requests R appear on the left side of both x and x$'$ although possibly in different orders. By the argument presented above for Lemma 1 using $\mathcal{S}$-commutativity, it is proved that the post-state of the set of requests R in x and x$'$ is the same. We showed above that r^* is permissible in the post-state of R in x$'$. Thus, r^* is permissible in the post-state of R in x as well. In other words, r^* is permissible at the pre-state of the set of requests R'' in x. In Fig. 4.(b), r^* is permissible in σ'_2, the post-state of r_4 in x.

The requests R'' precede r^* in xs(n) but succeed it in xs(n'). Therefore, by the $\mathcal{P}$-conflict-synchronization condition, each request in R'' $\mathcal{P}$-R-commutes with r^*. In Fig. 4.(a), r^* $\mathcal{P}$-R-commute with r''_1 and r''_2. We proved above that the request r^* is permissible at the pre-state of R'' in x. By induction from left to right in x, using the $\mathcal{P}$-R-commutativity, r^* is permissible at the post-state of each request r'' in R''. Therefore, r^* is permissible at its pre-state σ in x. In Fig. 4.(b), r^* is permissible at the states σ_1 and σ. Therefore, r^* is permissible at its pre-state in xs(n). ■

We note that conflict-synchronization is stronger than dependency-preservation. If a request r both conflicts with and depends on r', it is sufficient to synchronize r with r' and its dependencies to r' do not need to be tracked.

4 STATIC ANALYSIS

In the previous section, we defined conflict and dependency relations between methods. In this section, we recast the definitions as a static analysis that calculates these relations. The user specifies an object $\langle \Sigma, \mathcal{I}, \mathcal{M} \rangle$ where Σ is the state type, $\mathcal{I}$ is the invariant and $\mathcal{M}$ is the set of methods. Given the object, Fig. 5 presents two functions ConflictRel() and DepRel() that calculate the two relations. We consider each one in turn and apply them to our running example.

The function ConflictRel() returns the conflict relation as a mapping from pairs of methods $\mathcal{M} \times \mathcal{M}$ to boolean $\mathbb{B}$. It first calculates the $\mathcal{S}$-commutativity relation in the variable SCom (at lines C_6-C_7). Following Def. 11, for every pair of methods m_1 and m_2, $\mathsf{SCom}(m_1, m_2)$ is true iff the following assertion is valid: for every pre-state σ, argument a_1 of m_1 and argument a_2 for m_2, the post-states of applying the two calls $m_1(a_1)$ and $m_2(a_2)$ on σ in the two different orders are equal. We use the notation $\vdash \mathcal{A}$ to represent whether the assertion $\mathcal{A}$ is valid. To check the validity of an assertion, we use SMT solvers to check the satisfiability of its negation.

For example, Fig. 1.(b) shows that the two methods addCourse and enroll $\mathcal{S}$-commute. Let us see how this is calculated. To calculate the value of $\mathsf{SCom}(\mathsf{addCourse}, \mathsf{enroll})$, the assertion in line C_7 is instantiated to the following assertion. (The pre-state σ is expanded to $\langle ss, cs, es \rangle$, the argument of addCourse is c and the arguments of enroll are s and c'.)

$$\begin{aligned} \vdash \forall ss, cs, es, c, s, c'.\ & \mathsf{update}(\mathsf{enroll}(s, c'))(\mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es \rangle)) = \\ & \mathsf{update}(\mathsf{addCourse}(c))(\mathsf{update}(\mathsf{enroll}(s, c'))(\langle ss, cs, es \rangle)) \end{aligned} \tag{1}$$

Based on the object definition in Fig. 1.(a), the two expressions can be simplified as follows:

$$\begin{array}{ll} \text{Left exp:} & \mathsf{update}(\mathsf{enroll}(s, c'))(\mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es \rangle)) \ = \\ & \mathsf{update}(\mathsf{enroll}(s, c'))(\langle ss, cs \cup \{c\}, es \rangle) \ = \ \langle ss, cs \cup \{c\}, es \cup \{\langle s, c' \rangle\} \rangle \\ \text{Right exp:} & \mathsf{update}(\mathsf{addCourse}(c))(\mathsf{update}(\mathsf{enroll}(s, c'))(\langle ss, cs, es \rangle)) \ = \\ & \mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es \cup \{\langle s, c' \rangle\} \rangle) \ = \ \langle ss, cs \cup \{c\}, es \cup \{\langle s, c' \rangle\} \rangle \end{array} \tag{2}$$

The two expressions are equal; thus, the assertion is valid and the two methods $\mathcal{S}$-commute.

Similar to $\mathcal{S}$-commutativity, the other relations are calculated by a validity check for their definitions. In summary, the ConflictRel() function calculates the invariant-sufficiency relation (Def. 12) in the variable ISuff (at C_8-C_9) and the $\mathcal{P}$-R-commutativity relation (Def. 13) in the variable PRCom (at C_{10}-C_{11}). They are used to calculate the $\mathcal{P}$-concur relation (Def. 14) in the variable PConcur (at line C_{12}). Then, the concur relation (Def. 15) for a pair of methods is calculated in the variable Concur as the conjunct of $\mathcal{S}$-commutativity and $\mathcal{P}$-concur of the method pair with respect to each other (at C_{13}-C_{14}). (We note that $\mathcal{S}$-commutativity is symmetric.) Finally, the conflict relation (Def. 15) is calculated as the negation of the concur relation in the variable Conflict and returned (at C_{15}-C_{16}). These steps calculate the sub-figures (b) to (e) of Fig. 1 in order.

The function DepRel() calculates the dependency relation. It first calculates invariant-sufficiency (Def. 12) in the variable ISuff (at lines D_5-D_6) and $\mathcal{P}$-L-commutativity (Def. 17) in the variable PLCom (at D_7-D_8). They are used to calculate the independence relation (Def. 18) in the variable Indep (at D_9-D_{10}). Finally, the dependence relation (Def. 18) is calculated as the negation of the independence relation in the variable Dep and returned (at D_{11}-C_{12}).

Fig. 1.(f) and (g) show that enroll is dependent on addCourse. Let us see how this is calculated. We show that enroll is not invariant-sufficient and does not $\mathcal{P}$-L-commute with addCourse either. First, we show that the method enroll is not invariant-sufficient. Intuitively, even if the invariant holds in the pre-state of enroll, it does not trivially hold in its post-state. The invariant-sufficiency assertion that is checked at D_6 is instantiated to the following assertion: (The pre-state σ is expanded to $\langle ss, cs, es \rangle$ and the arguments of enroll are s and c.)

$$\vdash \forall ss, cs, es, s, c.\ \mathcal{I}(\langle ss, cs, es \rangle) \rightarrow \mathcal{P}(\langle ss, cs, es \rangle, \mathsf{enroll}(s, c)) \tag{3}$$

After unrolling $\mathcal{P}$, the conclusion of the implication includes the following conjunct

$$\begin{aligned} & \mathcal{I}(\mathsf{update}(\mathsf{enroll}(s, c))(\langle ss, cs, es \rangle)) \ = \ \mathcal{I}(\langle ss, cs, es \cup \{\langle s, c \rangle\} \rangle) \ = \\ & \mathsf{refIntegrity}(es \cup \{\langle s, c \rangle\}, \mathsf{esid}, ss, \mathsf{sid}) \ \wedge \ \mathsf{refIntegrity}(es \cup \{\langle s, c \rangle\}, \mathsf{ecid}, cs, \mathsf{cid}) \end{aligned} \tag{4}$$

According to the definition of referential integrity in the caption of Fig. 1, the first conjunct is expanded to the following assertion:

$$\forall r.\ \ r \in es \cup \{\langle s, c\rangle\} \rightarrow \exists r'.\ r' \in ss \land \mathsf{esid}(r) = \mathsf{sid}(r') \tag{5}$$

We note that s is an unconstrained universally quantified variable in Eq. 3, the original invariant-sufficiency assertion. Therefore, to falsify that assertion, the variable s can be instantiated with any student value. Enrolling any student s that is not already in ss violates the above referential integrity property and leads to a counter-example for validity for Eq. 3. Intuitively, enrolling a student that is not already in the students relation violates integrity. Hence, the method enroll is not invariant-sufficient.

Next, we show that enroll does not $\mathcal{P}$-L-commute with addCourse. Intuitively, the enroll method does not preserve its permissibility if it is moved left before a preceding addCourse. The assertion in line D_8 is instantiated to the following assertion. (The pre-state σ is expanded to $\langle ss, cs, es\rangle$, the argument of addCourse is c and the arguments of enroll are s and c'.)

$$\begin{array}{l}\vdash \forall ss, cs, es, c, s, c'. \\ \ \ \mathcal{P}(\mathsf{update}(\mathsf{addCourse}(c))(\langle ss, cs, es\rangle), \mathsf{enroll}(s, c')) \rightarrow \mathcal{P}(\langle ss, cs, es\rangle, \mathsf{enroll}(s, c'))\end{array} \tag{6}$$

The counter-example is when $c = c'$, that is the same course is added and enrolled, and $c \notin cs$, that is c is not already an existing course. After expansion and removing the trivially valid guard assertions, we have

$$\mathcal{I}(\langle ss, cs \cup \{c\}, es \cup \{s, c\}\rangle) \rightarrow \mathcal{I}(\langle ss, cs, es \cup \{s, c\}\rangle) \tag{7}$$

Expanding the conclusion of the implication results in the following conjunct:

$$\forall r.\ r \in es \cup \{\langle s, c\rangle\} \rightarrow \exists r'.\ r' \in cs \land \mathsf{ecid}(r) = \mathsf{cid}(r') \tag{8}$$

This assertion is invalid. For $r = \langle s, c\rangle$, the conclusion never holds as $c \notin cs$. This makes a counter-example for the $\mathcal{P}$-L-commutativity assertion. Thus, enroll does not $\mathcal{P}$-L-commute with addCourse. A call on enroll is dependent on the preceding addCourse call.

We note that the premise of the implication in Eq. 7 does not refute the choice that $c \notin cs$. In the premise of Eq. 7, the integrity of the enrolment relation $es \cup \{\langle s, c\rangle\}$ for the course c may hold only because c was just added and resulted in the course relation $cs \cup \{c\}$ and not because it already existed in cs.

We note that since local permissibility is a condition of a well-coordinated replicated execution, every call is permissible in its originating node. Therefore, every call in all the conditions above can be additionally assumed to be permissible in a fresh state (unrelated to the other state variables in the condition). We elided this permissibility condition for brevity. Permissibility even in an unrelated state can provide useful information. In particular, the validity of the guard of the call can provide conditions on the arguments of the call that are independent of the state.

5 USE-CASES

We now present two use-cases. (All of our use-cases are available in the appendix § 2.)

Fig. 6.(a) represents the Auction use-case that we have adopted from CISE [Gotsman et al. 2016]. Users can place bids and then the auction can be closed to declare the winner. The state Σ of the object is the record of the set of current bids bs, and the option value w that is either some winning bid or none $\perp$ when the auction is still open. The integrity invariant $\mathcal{I}$ is that if the auction is closed, then the winning bid is the maximum of the non-empty set of bids. Auction offers three methods: place, close and query. While the auction is open, the method place can place a bid b. The method close closes the bid by picking the maximum bid. The method query returns the current state of the auction. It is notable that in the guard of close, we do not need to repeat the condition

```
Class Auction
  $\Sigma := \langle bs : \mathsf{Set\ Int},\ w : \mathsf{Option\ Int} \rangle$
  $\mathcal{I} := \lambda\ \langle bs, w \rangle.$
      $w \neq \bot \rightarrow (bs \neq \emptyset\ \land\ w = \mathsf{some}(\mathsf{max}(bs)))$
  $\mathsf{place}(b) := \lambda\ \langle bs, w \rangle.$
      $\langle w = \bot,\quad \langle bs \cup \{b\},\ w \rangle,\quad \bot \rangle$
  $\mathsf{close} := \lambda\ \langle bs, w \rangle.$
      $\langle w = \bot,\quad \langle bs,\ \mathsf{some}(\mathsf{max}(bs)) \rangle,\quad \bot \rangle$
  $\mathsf{query} := \lambda\ \sigma.\ \langle \mathbb{T},\ \sigma,\ \sigma \rangle$
```

(a) User Specification

```
Class 2PSet
  $\Sigma := \langle \mathsf{Set}, \mathsf{Set} \rangle$
  $\mathcal{I} := \mathbb{T}$
  $\mathsf{add}(e) := \lambda\ \langle A, R \rangle.$
      $\langle \mathbb{T},\quad \langle A \cup \{e\}, R \rangle,\quad \bot \rangle$
  $\mathsf{remove}(e) := \lambda\ \langle A, R \rangle.$
      $\langle \mathbb{T},\quad \langle A, R \cup \{e\} \rangle,\quad \bot \rangle$
  $\mathsf{contains}(e) := \lambda\ \langle A, R \rangle.$
      $\langle \mathbb{T},\quad \langle A, R \rangle,\quad e \in A \setminus R \rangle$
```

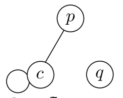
(f) User Specification

	p	c	q
p	✓	×	✓
c	×	✓	✓
q	✓	✓	✓

(b) $\mathcal{S}$-commute

	p	c	q
p	✓	×	✓
c	✓	×	✓
q	✓	✓	✓

(c) $\mathcal{P}$-concur



(d) Conflict graph

	p	c	q
p	✓	✓	✓
c	×	✓	✓
q	✓	✓	✓

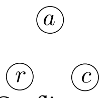
(e) Independent

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(g) $\mathcal{S}$-commute

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(h) $\mathcal{P}$-concur



(i) Conflict graph

	a	r	c
a	✓	✓	✓
r	✓	✓	✓
c	✓	✓	✓

(j) Independent

Fig. 6. Auction and Two Phase Set Use-cases. The conflict graph in (d) is obtained from (b) and (c).

that the bid set should be non-empty. This condition is declared in the invariant. If a call on close violates the invariant, the call is not permissible and is aborted. In general, the user does not need to restate the invariant as guards. The guard needs to only specify the semantic preconditions of the method. Thus, our specifications are simpler than previous work [Gotsman et al. 2016]. As an example of semantic preconditions, the execution of a close call on an auction is meaningful only if the auction is not already closed although it does not violate the invariant. Similarly, placing a bid is meaningful only when the auction is not closed even if the bid is less than the already decided winner which does not violate the integrity of the auction.

Fig. 6.(b) shows that the place and close methods $\mathcal{S}$-conflict. a call on place can execute either before or after a call on close. In the former, the close method gets to see the new bid that might be the largest. However, in the latter, the new bid is missed. Therefore, the two executions can diverge. As Fig. 6.(c) shows, the place and close methods $\mathcal{P}$-conflict with the close method. The methods place and close are not invariant-sufficient. Their guards require the auction to be open that is not implied by the invariant. If a call on place is pushed after a call on close, the call on place can violate the invariant as it can place a bid larger than the already decided winner. If a call on close is pushed after another call on close, its guard does not hold after the move. As Fig. 6.(e) shows, a call on close is dependent on a preceding call on place. The preceding place call can be placing the only bid and if it is removed, the close call gets an empty auction to close that violates the invariant.

Fig. 6.(f) shows the 2PSet (two-phase set) use-case that we have adopted from CRDTs [Shapiro et al. 2011]. Classical sets $\mathcal{S}$-conflict on adding and removing elements. The two orders do not agree on the final set. However, this is only when the two calls are on the same element. A set with a known finite domain can avoid conflicts and synchronization for unequal elements. We present the finite set in the appendix § 2 where we consider add or remove calls on each element separately leading to a larger table with fewer conflicts. In contrast, 2PSet avoids conflicts by changing the set semantics: once an element is removed, it cannot be added again. As Fig. 6.(f) shows, it uses two sets to store added and removed elements and the abstract state of the set is the added set minus the removed set. Therefore, the two orders of adding and removing an element result in the same set: the element is considered to be removed. As Fig. 6.(g)-(j) shows, the methods of 2PSet concur and are independent. Thus, 2PSet methods can execute without any coordination. In the appendix, we apply 2PSet to the Courseware use-case to reduce set conflicts.

6 PROTOCOLS

In the previous sections, we presented how the conflict and dependency relations of a given object are calculated. In this section, we present two concrete protocols that use these relations and implement the well-coordination conditions. The protocols are parametric and instantiated with the object and its conflict and dependency relations. The first protocol is non-blocking. Crash of a replica does not prevent other replicas from making progress. The second protocol is blocking. In return, it can further avoid synchronization. For a pair of conflicting methods, the protocol can push synchronization to only one of them and the other method can execute without synchronization.

In the next two subsections, we focus on synchronization of conflicting methods (and assume that methods are independent). We consider dependencies in the third subsection.

Each protocol declares the request events that it inputs and the response events that it outputs. It also declares the state that it stores at every node. It may include an initialization method that is called once at the beginning of the execution at each node. A protocol may declare and use other protocols. It defines methods for requests from the client and responses from the used protocols. A method may be guarded by a condition. Such a method accepts events only when the condition is satisfied; otherwise, the processing of the event is postponed. In the body of the methods, a protocol may issue responses to its client or issue requests to the used protocols.

6.1 Non-blocking Synchronization Protocol

In this subsection, we present a non-blocking protocol to synchronize conflicting calls.

Protocol Idea. A subset of the vertices of a graph is a clique iff any of its distinct pair of vertices are adjacent. A clique is maximal if it is not a subset of a larger clique. There are known algorithms [Bron and Kerbosch 1973; Tsukiyama et al. 1977] that list the set of maximal cliques of a graph.

The methods of a clique of the conflict graph have to all synchronize with each other. The idea is to synchronize only the methods of each maximal clique with each other to minimize synchronization. We use the total-order broadcast (TOB) protocol that employs consensus to deliver messages in the same total order to all nodes [Cachin et al. 2011]. Let Cl denote the set of maximal cliques of the conflict graph. For each maximal clique $cl \in \mathsf{Cl}$, we use a TOB instance $tob(cl)$. Calls on conflicting methods in a maximal clique are broadcast to a TOB instance and delivered with the same total order to all nodes. (A single node (without a loop) is considered a clique but does not need synchronization. Thus, before calculating the maximal cliques, we remove all the single nodes without loops from the conflict graph.) As we saw in Fig. 3.(a) for the delete call d, a call c of a method m that is a member of multiple maximal cliques cls should be totally ordered with respect to calls of each of those cliques. We broadcast the call c to every $tob(cl)$ where $cl \in cls$ and execute c only when it is ordered and delivered by all of them. To execute calls in the delivery order from TOBs, we maintain a queue $q(cl)$ for each TOB instance $tob(cl)$. Method calls that are delivered by a TOB instance $tob(cl)$ are enqueued to its corresponding queue $q(cl)$. If cls is the set of maximal cliques containing a method m, a call on m can be executed once it appears at the head of the queues $q(cl)$ for each $cl \in cls$. The call is then dequeued from the queues and executed. Thus, the execution order of calls at every replica is an extension of the delivery order of each of the TOBs. Therefore, calls to conflicting methods have the same execution order across replicas.

However, deadlocks can happen if the TOB instances are not properly coordinated. Consider two method calls c and c' of a method m that is a member of two cliques cl and cl'. If c and c' are simply broadcast to $tob(cl)$ and $tob(cl')$, c may precede c' in the total order of $tob(cl)$ and succeed it in the

```
      NonBlockingRepObject
          request: call(C)
          response: ret(C, V)
                    aborted(C)
      Params:
          cliques: M → List[Cl]
      Using:
          $rb$: ReliableBroadcast
          $mtob$: Cl → MultiTotalOrderBroadcast
      State:
          $\sigma$: $\Sigma = \sigma_0$
          $q$: Cl → Queue[C] = Cl ↦ ∅
$R_0$   request (call($c$))
$R_1$       $m$ ← method($c$)
$R_2$       $cls$ ← cliques($m$)
$R_3$       if ($cls = \emptyset$)
$R_4$           issue request ($rb$, broadcast($c$))
$R_5$       else
$R_6$           $cl$ ← head($cls$)
$R_7$           issue request ($mtob(cl)$, broadcast($c, \bot$))
$N_0$   response ($rb$, deliver($c$))
$N_1$       exec($c$)
$I_1$   response ($mtob(cl)$, deliver($c$))
$I_2$       enq($q(cl), c$)
$I_3$       $m$ ← method($c$)
$I_4$       $cls$ ← cliques($m$)
$I_5$       if ($cl$ = last($cls$))
$I_6$           check($c$)
$I_7$       else
$I_8$           $cl'$ ← next($cls, cl$)
$I_9$           issue request ($mtob(cl')$, broadcast($c, cl$))
$C_0$   fun check($c$)
$C_1$       $m$ ← method($c$)
$C_2$       $cls$ ← cliques($m$)
$C_3$       if (forall $cl \in cls$. head($q(cl)$) = $c$)
$C_4$           foreach($cl \in cls$) $q(cl)$.deq()
$C_5$           exec($c$)
$C_6$           checkQs()
$C_7$           return true
$C_8$       else
$C_9$           return false
$Q_1$   fun checkQs()
$Q_2$       foreach($cl \in$ Cl, $q(cl) \neq \emptyset$)
$Q_3$           $c$ ← head($q(cl)$)
$Q_4$           if (check($c$)) return
$E_1$   fun exec($c$)
$E_2$       if (guard($c$)($\sigma$) $\wedge$ $\mathcal{I}$(update($c$)($\sigma$)))
$E_3$           $\sigma$ ← update($c$)($\sigma$)
$E_4$           $v$ ← retv($c$)($\sigma$)
$E_5$           issue response ret($c, v$)
$E_6$       else
$E_7$           issue response aborted($c$)
```

Fig. 7. Non-blocking Synchronization Protocol. C and V are call and return value respectively

total order of $tob(cl')$. Thus, c' cannot appear at the head of the queue $q(cl)$ and waits for c and symmetrically c cannot appear at the head of the queue $q(cl')$ and waits for c'. As a result, c and c' and all later calls in $q(cl)$ and $q(cl')$ will be blocked at all replicas. To prevent deadlocks, firstly, we statically order the maximal cliques Cl and always send a message to TOB instances $tob(cl)$ in the order of their corresponding cliques cl. Secondly, we ensure that if a message is ordered before another by a TOB instance then the next TOB instance respects this order. To this end, we present and use a particular kind of total-order broadcast that respects given total orders on subsets of messages. We call it the multi-total-order broadcast (MTOB) protocol.

In the multi-total-order broadcast (MTOB) protocol, the messages are divided to multiple disjoint subsets called message classes. Each class is associated with a total order. The user broadcasts each message together with its class identifier. She should also broadcast messages of a class in the total order of that class. The protocol delivers messages in a total order that respects (i.e. is an extension of) the order of each message class.

We use MTOB as follows. We define a class as the set of calls of the methods of a clique. As mentioned above, a call is sent to the MTOB instances in a statically-determined order. For example, in the example above, we assume that $mtob(cl)$ is before $mtob(cl')$ in the static order. Assume that c is delivered before c' by $mtob(cl)$. We broadcast c and c' in order to $mtob(cl')$ with class cl. As the order of messages in class cl is preserved in the delivery order of $mtob(cl')$, c will be delivered before c' by $mtob(cl')$ as well and the deadlock mentioned above cannot happen.

We first present the main protocol and then the multi-total-order-broadcast protocol. They use the classical reliable broadcast and consensus protocols [Cachin et al. 2011].

Main Protocol. The non-blocking protocol is presented in Fig. 7. The requests to the protocol are call(c) to execute a method call c on the replicated object. In response, the protocol issues the

```
     MultiTotalOrderBroadcast
         request: broadcast(M, C)
         response: deliver(M)
     Using:
         $rb$: ReliableBroadcast
         $cs$: $R \rightarrow$ Consensus
     State:
         $p$: Set[M × C × Int] = $\emptyset$       Pending
         $d$: Set[M × C × Int] = $\emptyset$       Delivered
         $r$: Int = 0                    Round
         $rank$: $C \rightarrow$ Int = $C \mapsto 0$     Rank
$I_0$  init()
$I_1$      issue request ($cs(0)$, propose($\emptyset$))
$R_0$  request (broadcast($m, c$))
$R_1$      if ($c \neq \bot$)
$R_2$          $rank(c) \leftarrow rank(c) + 1$
$R_3$          issue request ($rb$, broadcast($m, c, rank(c)$))
$R_4$      else
$R_5$          issue request ($rb$, broadcast($m, \bot, 0$))
$D_0$  response ($rb$, deliver($m, c, i$))
$D_1$      if (($m, c, i$) $\notin d$)
$D_2$          $p \leftarrow p \cup \{(m, c, i)\}$
$C_0$  response ($cs(r')$, decide($d'$)) if ($r' = r$)
$C_1$      foreach(($m, c, i$) $\in$ sort($d'$))
$C_2$          issue response deliver($m$)
$C_3$      $d \leftarrow d \cup d'$
$C_4$      $p \leftarrow p \setminus d'$
$C_5$      $r \leftarrow r + 1$
$C_6$      issue request ($cs(r)$, propose(proposal()))
$P_0$  fun proposal()
$P_1$      $\{(m, c, i) \mid (m, c, i) \in p \wedge$
$P_2$          $\forall i'. \; 0 < i' < i \rightarrow \exists m'. \; (m', c, i') \in d\}$
```

Fig. 8. Multi-Total-Order Broadcast Protocol. M and C are message and class types respectively.

response ret(c, v) to return the value v as the result of the call c or aborted(c) to indicate that the call c could not be executed without the violation of the invariant and is aborted. The parameter to the protocol is the map cliques. It maps each method in the set of methods M to a list of maximal cliques Cl that the method belongs to. As explained earlier, the set of maximal cliques is calculated and statically sorted to a total order. To prevent deadlocks, every list in the range of cliques is consistent with this total order. A call on a method m is sent to the TOB instances of the cliques cliques(m) in order. The protocol uses two protocols: reliable broadcast rb and a multi-total-order broadcast per clique $mtob$. Among other properties, the reliable broadcast guarantees that if a message is delivered by a correct node, then it is eventually delivered by every correct node. In addition to this guarantee, as previously mentioned, MTOB protocol guarantees that messages are delivered in a total order that is an extension of the order of each message class. Each replica stores the following state: the state σ of the user-defined object, and the queues q, one per maximal clique.

On the invocation of the request call(c) to execute the call c (at R_0), the protocol finds the method m of c (at R_1) and the set of cliques cls that m belongs to (at R_2). If the set of cliques is empty, no synchronization is needed and the request is sent using the reliable broadcast rb (at R_3-R_4). Otherwise, the request is sent using the MTOB instance $mtob(cl)$ of the first clique cl in cls. As this is the first broadcast, the call can be arbitrarily ordered and no class ($\bot$) is passed as the class (at R_5-R_7). When a call c is delivered by the reliable broadcast rb (at N_0), as no further synchronization is required, it is executed (at N_1). When a call c is delivered by an MTOB instance $mtob(cl)$ (at I_1), we enqueue it to the corresponding queue $q(cl)$ (at I_2), and get the list of cliques cls of the method (at I_3-I_4). If the current clique is the last one in the list (at I_5), we check if the call can be executed (at I_6). Otherwise, we send c to the next MTOB instance $mtob(cl')$. The call is broadcast together with the previous clique cl as the class (at I_7-I_9). A call is ready to be executed if it appears at the head of all the queues of the cliques that the method belongs to (at C_0-C_3). A call c that is ready is dequeued from the queues (at C_4) and executed (at C_5). Then, the queues are checked for next calls that might be ready to execute (at C_6). To check the queues (at Q_1), the call at the head of every queue is checked. Checking is repeated if a call is executed (at Q_2-Q_4 and C_5-C_9). To execute a call (at E_1), it is checked that it is locally permissible i.e. its guard is satisfied and applying it does not violate the invariant (at E_2). If the check is passed, the updated state is stored, the return value v is calculated (at E_3-E_4), and a return response is issued with v (at E_5). Otherwise, an abort response is issued (at E_6-E_7). As pairs of conflicting methods are synchronized and methods are independent, a call is permissible in one replica if and only if it is permissible in another.

The protocol is non-blocking: if a quorum (majority) of nodes are correct (not faulty), every request for a call will eventually get a response. The call is first broadcast to the *rb* or an *mtob*. Both will eventually deliver the call. (We will show this property for MTOB with a quorum of correct nodes.) In the former case, the call is executed on arrival. In the latter case, it is put in the corresponding queue and may be broadcast to the next *mtob*. As we explained above, each MTOB preserves the delivery order of the previous MTOBs; thus, two calls can appear in two queues only in the same order and cannot cause a deadlock. Calls eventually arrive at the head of the queues, are dequeued and executed.

Multi-Total-Order Broadcast. The multi-total-order broadcast (MTOB) protocol is presented in Fig. 8. The protocol accepts requests to broadcast a message m given its class c. (A message can belong to no class $\perp$. These messages are assumed to be unique.) MTOB delivers messages to every node with the same order and this order respects the order of all message classes. The idea is to have rounds of consensus to agree on the messages to deliver. In each round, nodes propose their current messages for consensus. When the consensus protocol issues the decision response with a set of messages, they are locally sorted using a deterministic sort algorithm and delivered. To respect the order of message classes, a message is proposed only if all the messages before it in the class are already delivered. Starvation of a node and its messages in the case that its proposal is repeatedly not chosen is prevented as follows. MTOB uses a reliable broadcast protocol. Upon a broadcast request for a message, it is first broadcast with the reliable broadcast protocol to other nodes. Thus, the message will be in the proposal of other nodes and will be eventually chosen.

MTOB uses the reliable broadcast protocol *rb* and an instance sequence of the consensus protocol *cs*. MTOB proceeds in rounds R and uses an instance of consensus in each round. It stores the set of pending messages p, the set of delivered messages d, the number of the current round r, and the rank of the last delivered message for each class *rank*.

The rounds of consensus are kick-started in the initialization function (at I_0-I_1). Upon an MTOB request to broadcast a message, if it belongs to a class (at R_1), the rank for the class is incremented (at R_2) and it is broadcast using the reliable broadcast *rb* (at R_3). Otherwise, the message is broadcast with no class and zero rank (at R_4). When *rb* delivers a message (at D_0), if it is not already delivered (at D_1), it is added to the pending set (at D_2). Once the decision of the current round is received (at C_0), its messages are sorted and delivered (at C_1-C_2) and added to the delivered set and removed from the pending set (at C_3-C_4). Then, the node enters the next round and proposes in it (at C_5-C_6). The proposal is the largest subset of the pending messages m such that all the messages before m in its class are already delivered (at P_0-P_2). This condition ensures that the order of each class is preserved. It is notable that as the messages with no class are added to the pending set with rank 0, they always satisfy the proposal condition. We elide the optimizations to the appendix § 4.

It is assumed that a quorum (majority) of nodes are correct. Let us explain why a message broadcast by a correct node is eventually delivered to every correct node. We consider a message of a class and by induction assume that previous messages of the class are eventually delivered. If the message has been and will be in the decided set of a round, it is or will be eventually delivered by all correct nodes. Otherwise, we assume that it is never in a decided set and thus never in a delivered set. The message is first broadcast using *rb*. Thus, it is eventually delivered by *rb* and as it is not in the delivered sets, it will be added to the pending set of all correct nodes. As the previous messages in the class are eventually delivered, the message will eventually be in the proposed set of all correct nodes. With a quorum of correct nodes, the consensus protocol guarantees eventual decision. Thus, the message will eventually be in the decided set and delivered.

```
     BlockingRepObject
       request: call(C)
       response: ret(C, V) | aborted(C)
     Params:
       conflict: M → Set[M]
       cover: Set[M]
     Using:
       rb: ReliableBroadcast
       tob: M → TotalOrderBroadcast
     State:
       $\sigma$: $\Sigma = \sigma_0$
       $b$: M → Int = M ↦ 0
       $xed$: M → Set[C] = M ↦ ∅
       $act$: M → $\mathbb{B}$ = M ↦ false
       $cnt$: C → Int = C ↦ 0
$R_0$  request (call($c$))
$R_1$    $m \leftarrow$ method($c$)
$R_2$    if ($m \notin$ cover)
$R_3$      issue request ($rb$, broadcast(nsync($c$)))
$R_4$    else
$R_5$      if ($m \in$ conflict($m$))
$R_6$        issue request ($tob(m)$, broadcast(sync($c$)))
$R_7$      else
$R_8$        issue request ($rb$, broadcast(sync($c$)))
$N_0$  response ($rb$, deliver(nsync($c$))) if $b$(method($c$)) = 0
$N_1$    exec($c$)
$C_0$  response ($tob(m)$, deliver(sync($c$))) if $\neg act(m)$
$C_1$    $act(m) \leftarrow$ true
$C_2$    blockAndUpdate($c$)
$C_3$  response ($rb$, deliver(sync($c$)))
$C_4$    blockAndUpdate($c$)
$B_1$  fun blockAndUpdate($c$) where $m \leftarrow$ method($c$)
$B_2$    foreach($m' \in$ conflict($m$))  $b(m') \leftarrow b(m') + 1$
$B_3$    $cs \leftarrow xed$ | conflict($m$)
$B_4$    issue request ($rb$, broadcast(update($c, cs$)))
$U_0$  response ($rb$, deliver(update($c, cs$)))
$U_1$    foreach($c' \in cs$)  exec($c'$)
$U_2$    $cnt(c) \leftarrow cnt(c) + 1$
$U_3$    if ($cnt(c) = \mathcal{N}$)
$U_4$      exec($c$)
$U_5$      foreach($m' \in$ conflict($m$))  $b(m') \leftarrow b(m') - 1$
$U_6$      $act(m) \leftarrow$ false
$E_0$  fun exec($c$)
$E_1$    if (guard($c$)($\sigma$) $\wedge$ $\mathcal{I}$(update($c$)($\sigma$)))
$E_2$      $\sigma \leftarrow$ update($c$)($\sigma$)
$E_3$      $v \leftarrow$ retv($c$)($\sigma$)
$E_4$      issue response ret($c, v$)
$E_5$      add($xed$(method($c$)), $c$)
$E_6$    else
$E_7$      issue response aborted($c$)
```

Fig. 9. Blocking Synchronization Protocol

6.2 Blocking Synchronization Protocol

The previous protocol requires both calls of a conflicting pair to participate in synchronization. In this section, we introduce a blocking protocol. As we saw in Fig. 3.(b) for the add a and enroll e calls, this protocol can make one of the two calls execute without synchronization.

Protocol Idea. Consider two conflicting methods m and m', and two calls c on m and c' on m'. To let the call c execute without synchronization, the other call c' needs to reach out to other nodes, block the execution of calls on m at those nodes and then propagate previous calls on m from every node to other nodes. Then, c' can be executed at all nodes. At the end, the execution of calls on m is unblocked at all nodes. Therefore, the set of calls on m before each call on m' is the same across nodes. This means that the order of every pair of calls on m and m' is the same across nodes.

A vertex cover V' of a graph $\langle V, E\rangle$ is a subset of the vertices V such that every edge in E has at least one endpoint in V'. A minimum vertex cover of a graph is a vertex cover of the smallest size. In a graph with weighted vertices, the weighted minimum vertex cover is a vertex cover of the smallest weight sum. Finding the (weighted) minimum vertex cover is a classical graph problem.

In the interest of avoiding synchronization, we find the minimum vertex cover of the conflict graph. Only the methods in the cover synchronize and the rest can execute without synchronization. To execute a method in the cover, the requesting node has to reach out to all nodes and block and solicit the conflicting methods. If the user calls a method more often than others or favors its responsiveness, she can assign a lower weight to that method and apply the weighted minimum vertex cover. Methods can be assigned weights inversely proportional to their call frequency. To enforce that a method becomes synchronization-free, infinity can be assigned to its weight.

Protocol. The blocking synchronisation protocol is presented in Fig. 9. It accepts requests to execute calls and in return issues responses with the return value or that the call is aborted. The parameters to the protocol are the map conflict that maps every method to its set of conflicting methods and a vertex cover, cover, of the conflict graph. The protocol uses two classical protocols: the reliable broadcast *rb* and a total-order broadcast per method *tob*. The protocol stores the following state at each node: the user-defined state of the object σ, a mapping b from each method to the number of times that it is blocked, a mapping *xed* from each method to the set of executed calls on that method, a mapping *act* from each method to whether there is an active execution of a call on the method, a mapping *cnt* from each call to the number of messages received for it.

Upon a request to execute a method call c (at R_0), if its method m is not a member of the cover (at R_1-R_2), it can be executed without synchronization. So, it is broadcast using *rb* as a non-synchronizing nsync call (at R_3). Otherwise, the call should synchronize with conflicting methods (at R_4) and it is broadcast as a synchronizing sync call. If m has a self-loop in the conflict graph, then c should synchronize with other calls on m. To order calls on m, they are broadcast using the total-order-broadcast $tob(m)$ (at R_5-R_6). If m does not have a self-loop, c only needs to synchronize with calls on other methods. Thus, c is broadcast using the reliable broadcast *rb* (at R_7-R_8).

Upon receiving a non-synchronizing call that is not blocked (at N_0), it is executed (at N_1). A call on a blocked method should wait until it is unblocked. When a synchronizing call c on a method m is received from a total-order broadcast $tob(m)$, if the execution of another call on m is not active (at C_0), it is recorded that the execution of a call on m is active (at C_1). On the other hand, when a synchronizing call is received from the reliable broadcast *rb* (at C_3), it does not need to prevent other calls on m as m does not conflict with itself. In both cases (at C_2 and C_4), each method that conflicts with m is blocked (at B_2), and the calls on the conflicting methods that this node has executed are broadcast as an update to other nodes (at B_3-B_4). When an update arrives (at U_0), its calls are executed (at U_1) and the number of received updates for the call is incremented (at U_2). When an update from all nodes is received (at U_3), the call is executed (at U_4), the previously blocked methods for c are unblocked (at U_5), and it is recorded that the execution of a call on m is no longer active (at U_6). The execution of a call (at E_0-E_7) is similar to the previous protocol.

As mentioned earlier, this protocol brings more synchronization-freedom. However, either progress or consistency and convergence of nodes may be affected by crashes. Blocked operations are only unblocked when update messages are received from all the other nodes. If the update message from a node is not received, calls on the blocked methods cannot be executed. Either the network is slow or that node has crashed. If other nodes assume the former, the latter may be the case and they can never execute the blocked methods. On the other hand, if other nodes assume that the node has crashed, the network may be just slow. In particular, if a correct node n is mistakenly suspected while a synchronizing call c is being executed, consider that other nodes refrain from waiting for n, execute c and unblock conflicting methods before n blocks conflicting methods. Then, a node n' can execute a call c' on a conflicting method. The call c' can reach and execute at the suspected node n before it executes c. Thus, c' is after c at n' but before it at n. Therefore, the two conflicting method calls have different orders in different nodes. Further, c can become impermissible after c'. Thus, this can cause divergence and violation of integrity at node n.

6.3 Dependency-Tacking Protocol

In the presented synchronization protocols, we assumed that method calls were independent. However, as we saw in Fig. 2.(e), permissibility of a call at a node may be dependent on the preceding calls at that node; the call may not be permissible at other nodes.

We saw that method calls may or may not need to synchronize before execution. If a call did not need synchronization, it was simply broadcast and was immediately executed on arrival. For both the non-blocking protocol (Fig. 7) and the blocking protocol (Fig. 9) this was at N_1. However, if it has dependencies, they should be tracked at the originating node and broadcast together with the call. The receiving nodes should apply the call only after its dependencies are applied. On the other hand, some calls go through synchronization before execution. When synchronization is finished for a call c, it may or may not be permissible in different nodes. For the non-blocking protocol (Fig. 7), this is at C_5 and for the blocking protocol (Fig. 9), this is at U_4. If there is a node n that finds c permissible, every node can become permissible for c after n propagates the dependencies. The call c is aborted only if it is impermissible at every node. We use a protocol that is the inverse of the classical atomic commit protocol. The decision is abort if every replica votes for abort and is commit otherwise. Every node that finds c permissible votes for commit together with the dependencies of c and every node that finds it impermissible votes for abort. If a node receives the abort decision, it aborts the execution of c. If a node receives the commit decision, it waits for the dependencies. After the dependencies are applied, the call c is permissible and is executed. The detailed protocol is available in the appendix § 3.

7 IMPLEMENTATION

In this section, we describe the implementation of our synthesis tool, Hamsaz. The input to Hamsaz is the definition of an object that includes the state type and invariants on the state along with methods. Hamsaz synthesizes non-blocking and blocking replicated objects. It also outputs the baseline sequentially consistent replicated object. Hamsaz consists of two main parts: (1) determining the conflicts and dependencies and (2) instantiating the protocols.

Conflict and Dependency Analysis. We use the CVC4 [Barrett et al. 2011] SMT solver [Barrett et al. 2010] to decide the validity of concur and independence relations for pairs of methods. In particular, we use the theory of linear arithmetic, inductive datatypes, and more importantly, the theory of finite sets [Bansal et al. 2016] and the follow-up theory of finite relations [Meng et al. 2017] that is recently added to CVC4. Decidable fragments of set theory [Cantone et al. 2013] is an active area of research [Cantone and Zarba 2000; Kuncak and Rinard 2007; Suter et al. 2011].

To decide the validity of a condition, Hamsaz may decompose the invariant to conjuncts. As an example, consider whether $\mathsf{enroll}(s_1, c_1)$ $\mathcal{P}$-concurs with $\mathsf{enroll}(s_2, c_2)$ in the Courseware use-case presented in Fig. 1. We focus on the invariant $\mathsf{refIntegrity}(es, \mathsf{esid}, ss, \mathsf{sid})$; the other invariant is similar. The invariant is unrolled to $\forall e.\ e \in es \rightarrow \exists s.\ s \in ss \land \mathsf{esid}(e) = \mathsf{sid}(s)$. We decompose it to the following two conjuncts based on whether the referential integrity involves the enrolled student s_1: (1) $\forall e.\ e \in es \land \mathsf{esid}(e) = s_1 \rightarrow \exists s.\ s \in ss \land \mathsf{esid}(e) = \mathsf{sid}(s)$, (2) $\forall e.\ e \in es \land \mathsf{esid}(e) \neq s_1 \rightarrow \exists s.\ s \in ss \land \mathsf{esid}(e) = \mathsf{sid}(s)$. For the first one, the call $\mathsf{enroll}(s_1, c_1)$ $\mathcal{P}$-R-commutes with the call $\mathsf{enroll}(s_2, c_2)$. For the second one, the call $\mathsf{enroll}(s_1, c_1)$ is invariant-sufficient.

Protocols. We implemented the parametric protocols presented in § 6. Given the analysis results, we apply the graph optimizations and then instantiate the protocols with the optimization results. We implemented our protocols on top of APPIA [Carvalho and et. al. 2011], the accompanying toolkit of [Cachin et al. 2011]. It is a Java library of basic communication abstractions. We implemented our protocols on top of the basic broadcast, total-order broadcast and consensus protocols. We also implemented the sequentially consistent baseline. It uses a total-order broadcast instance to deliver calls to all nodes in the same order.

8 EVALUATION

We applied Hamsaz to a suite of use-cases to synthesize non-blocking and blocking replicated objects and compared their performance with the sequentially consistent baseline.

Use-cases. The use-cases are the following: (The uses-case and their concur and independence tables are available in the appendix § 2. Code snippets of a few uses-cases are available in the appendix § 6.) Counter: It can increment and decrement an integer value. NNCounter: The non-negative counter has the invariant that the counter value should be non-negative. Register: A register stores a value and provides methods to read and write it. BankAccount: The invariant is a non-negative balance. CSet: The classical set provides add, remove and contains methods. GSet: The grow-only set (adopted from [Shapiro et al. 2011]) provides adding (but not removing) an element contains methods. Both methods can execute without coordination. FDSet: A finite-domain set provides the classical set operations on a predefined finite set of elements. Thus, it can avoid coordination between calls on different elements. 2PSet (two-phase set) (adopted from [Shapiro et al. 2011]) and Auction (adopted from [Gotsman et al. 2016]) that we saw in Fig. 6.

$$
\begin{aligned}
&\mathsf{unique}(R, f) := \\
&\quad \forall r, r'.\ r \in R \land r' \in R \land f(r) = f(r') \rightarrow r = r' \\
&\mathsf{refIntegrity}(R, f, R', f') := \\
&\quad \forall r.\ r \in R \rightarrow \exists r'.\ r' \in R' \land f(r) = f'(r') \\
&\mathsf{rowIntegrity}(R, p) := \\
&\quad \forall r.\ r \in R \rightarrow p(r)
\end{aligned}
$$

Fig. 10. Relational Integrity Constrains

The suite includes relational use-cases as well. Relational integrity properties are specified using three predicates that we present in Fig. 10. The property $\mathsf{unique}(R, f)$ states that the values of the field f in the records of the relation R are unique. The property $\mathsf{refIntegrity}(R, f, R', f')$ states that for every record r in R, there exists a record r' in R' such that the field f of r is equal to the field f' of r'. The property $\mathsf{rowIntegrity}(R, p)$ states that every record of the relation R satisfies the predicate p. The relational uses cases are the following. Courseware: We saw the courseware use-case (adopted from [Gotsman et al. 2016]) in Fig. 1. It requires referential integrity for the student and course identifiers. 2PCourseware: It uses 2PSet to reduce conflicts in Courseware. Payroll: The payroll use-case (adopted from [Bailis et al. 2014]) stores employee and department relations. It requires uniqueness of employee identifiers, referential integrity for the department identifiers of employees, non-null values for employee names and non-negative salaries. It supports adding and removing employees and departments, and increasing and decreasing employee salaries. Tournament: The tournament use-case (adopted from [Balegas et al. 2015a]) stores players, tournaments, and enrolments. It requires uniqueness of player and tournament identifiers, referential integrity of player and tournament identifiers in enrolments, and that each player has a positive budget, each tournament has a size within a cap, and each active tournament has at least one player. It supports adding and removing players and tournaments, adding funds for a player, enrolling and disenrolling a player in a tournament, and beginning and ending a tournament.

Platform. The experiments are done on a cluster with 4 computing nodes. Each node has 2 AMD Opteron 6272 CPUs with a total 8 cores with 64GB ECC protected memory of RAM and a 40Gbps high-bandwidth low-latency InfiniBand network. The OS running on the cluster is CentOS 7.4 Linux x86_64 with the kernel version 3.10.0-862.3.2.el7. JDK is openjdk version 1.8.0_171 (OpenJDK 64-Bit Server VM build 25.171-b10, mixed mode). All nodes are connected to a Mellanox 18 port InfiniBand switch. Reported numbers are the arithmetic means of results from five repetitions.

Conflict and Dependency Analysis. The concur and independence conditions for Counter, NNCounter, Register and BankAccount use-cases all fall in the quantifier-free fragment of the theory of linear arithmetic. The conditions for CSet, GSet, FDSet and 2PSet all fall in the quantifier-free fragment of the theory of sets. However, the Auction use-case uses the max function. We specified the following two axioms for max and CVC4 could use them to decide the validity of the conditions. $\mathcal{A}_1 : \forall s, i.\ \ i \in s \rightarrow \mathsf{max}(s) \geq i$ and $\mathcal{A}_2 : \forall s.\ \ s \neq \emptyset \rightarrow \mathsf{max}(s) \in s$. The integrity properties of the relational uses-cases are encoded using quantifiers as presented in Fig. 10. The reason is that the current theory of sets in CVC4 does not support a complete set of relational operators. A set of operators is called complete if any relational algebra expression can be expressed by a combination of them. Selection (σ), projection (π), renaming (ρ), union ($\cup$), difference (\) and product ($\times$) are a complete set of operators. For example, a referential integrity refIntegrity(R, a, R', a') can be written as $\pi_a R \setminus \pi_{a'} R' = \emptyset$ using projection and difference and as Car$(R \bowtie_{a=a'} R') \geq$ Car(R) using join and cardinality. CVC4 supports difference and join but not projection and cardinality is a planned feature [Meng et al. 2017]. Despite using quantifiers, CVC4 can decide the validity of all conditions in our relational use-cases in less than a minute. We measured the time that Hamsaz takes to calculate the conditions and represent the results in Fig. 11. For each use-case, the table lists the number of methods, the number of invariants, the time to calculate $\mathcal{P}$-concur and $\mathcal{S}$-commute for the conflict relation, the time to calculate the independence relation and the total time. The table for all the use-cases is available in the appendix § 5.

Results. In this section, we compare the response time of our protocols with each other and the sequentially consistent (SC) baseline. The response time for a call is the time spent between the request and the response of the call. We conduct two experiments on the courseware use-case that we saw in Fig. 1 and the bank account use-case. In the bank account use-case, the withdraw method conflicts with itself and is dependent on deposit. The deposit and balance methods are conflict-free and independent. In the first experiment, we compare the response time of methods using different protocols. In the second one, we measure the effect of increasing the workload on the response time. In both experiments, we execute 500 calls evenly distributed on the methods.

Usecase	#M[1]	#$\mathcal{I}$[2]	$\mathcal{P}$[3]	$\mathcal{S}$[4]	Indep	Total
Bank	3	1	284	695	595	1574
Auction	3	2	405	921	571	1897
Courseware	5	4	950	3256	2597	6803
NNCounter	3	1	283	598	470	1351
Tournament	9	5	3482	25615	24146	53603

[1] The number of methods
[2] The number of invariants
[3] $\mathcal{P}$-concure time (ms)
[4] $\mathcal{S}$-commute time (ms)

Fig. 11. Analysis time

In the first experiment, we issue one call per millisecond and measure the average response time of the calls on each method. The results for the non-blocking and the SC protocols are shown in Fig. 12.(a) and for the blocking protocol are shown in Fig. 12.(b). We make this separation because the latter is two orders of magnitude more responsive than the former. The response time for SC is the same across methods as all methods use the same TOB instance. In the non-blocking protocol, the response time of the register and query methods is around a millisecond. The response time of these two methods is significantly less than that of the other methods because they can execute without coordination. The response time of the deleteCourse method is about two times that of addCourse and enroll methods because a deleteCourse call has to be ordered by two TOB instances while an addCourse or enroll call needs to be ordered by only one TOB instance. The enroll method is less responsive than the addCourse method because enroll has dependencies and needs to wait for them and addCourse does not. Both the SC and non-blocking protocols synchronize by TOB instances that rely on consensus. On the other hand, the blocking protocol avoids using TOB for the courseware use-case. Fig. 12.(b) shows that this avoidance significantly improves the response time. The two methods register and query execute without coordination. Calls on the deleteCourse method coordinate to block addCourse and enroll methods. Therefore, deleteCourse is less responsive than the other methods. Calls on addCourse and enroll methods may be blocked but when they are not, they execute without coordination.

We applied the first experiment to the bank account use-case as well. The results are shown in Fig. 12.(c). Similar to the courseware use-case, the SC protocol is the least responsive and uniform across all methods. In the other two protocols, the deposit and balance methods can execute without coordination in around a millisecond response time. Interestedly, the withdraw method exhibits almost the same response time in the blocking and non-blocking protocols. The reason is that withdraw conflicts with itself, and the blocking protocol uses a TOB to order withdraw calls (at

R_6 in Fig. 9). Thus, both the blocking and non-blocking protocols use TOB for the bank account use-case and the synchronization by the TOB dominates the execution time.

In the second experiment, we increase the workload from 10 to 800 calls per second and measure the average response time over all the calls. The results for the non-blocking and the SC protocols on the courseware use-case are shown in Fig. 12.(d). The result for the blocking protocol follow the same trend and are available in the appendix § 5. Similar to the first experiment, we make this separation because the latter is orders of magnitude more responsive than the former. As we increase the workload, the network transmits more messages and the protocol states grow that in turn affect the responsiveness. All the protocols get less responsive as the workload increases. The response time of the SC protocol grows faster as every operation goes through synchronization.

We observe that coordination specially synchronization can adversely affect the response time. The experiments suggest that our protocols can effectively avoid coordination to reduce the response time. They exhibit considerable improvement over the SC protocol. In particular, the blocking protocol is more responsive than the other protocols especially when no method conflicts with itself. However, as mentioned before, the blocking protocol may not progress in case of node crashes. On the other hand, the non-blocking protocol is less responsive but maintains progress.

9 RELATED WORKS

$\mathcal{I}$-confluence [Bailis et al. 2014] is a sufficient condition for invariant preservation of state-based replicated objects [Shapiro et al. 2011]. It states that if user operations and the merge operation are invariant preserving, then every execution is invariant preserving. In contrast to $\mathcal{I}$-confluence, well-coordination is a correctness condition for operation-based replicated objects [Shapiro et al. 2011]. Further, in addition to coordination-avoiding operations, it supports and reduces coordination for conflicting operations. A follow-up work, Blazes [Alvaro et al. 2017], applies a technique called sealing to replicated stream processing. It calculates deterministic results in the presence of non-deterministic reordering of messages. The idea is to split messages to windows and apply aggregate operations on them. Both the technique and its applications are distinct from ours.

Warranties [Liu et al. 2014] delay update operations for a limited time to preserve a state assertion on the distributed state. Thus, local computations can count on the assertion without coordination. However, in contrast to our approach, warranties are not automatically inferred and specifically improve the efficiency of read-dominated applications. Further, they preserve strong consistency rather than exploiting weak consistency. Homeostasis [Roy et al. 2015] targets invariants that span nodes of partially-replicated distributed stores. Each node maintains a condition called treaty on its local state and relies on the validity of other node treaties. The idea is that a change in the state of a node may preserve its treaty and not observationally change the execution of transactions in other nodes. Thus, coordination can be avoided. However, if a node violates its treaty, it synchronizes with other nodes and a new set of treaties are calculated and installed. In contrast, well-coordination targets fully-replicated stores, exploits weak consistency and guarantees convergence. Further, the analysis is static and the protocols do not calculate conditions at runtime.

Sieve [Li et al. 2014, 2012] defines a consistency model called RedBlue and applies static and dynamic analysis to determine whether an operation can be executed under causal consistency (blue) or needs strong consistency (red) to preserve the invariant. However, the analysis does not check that the result will indeed validate the invariant. In contrast, we prove the sufficiency of well-coordination. Further, causal consistency is the weakest possible notion in the RedBlue model while our model allows operations to execute with no synchronization and dependency.

Quelea [Sivaramakrishnan et al. 2015] lets the programmer declare consistency contracts for operations of a replicated object using primitive consistency relations such as visibility and session orders. It defines axiomatic semantics for consistency notions using the same primitives. It then

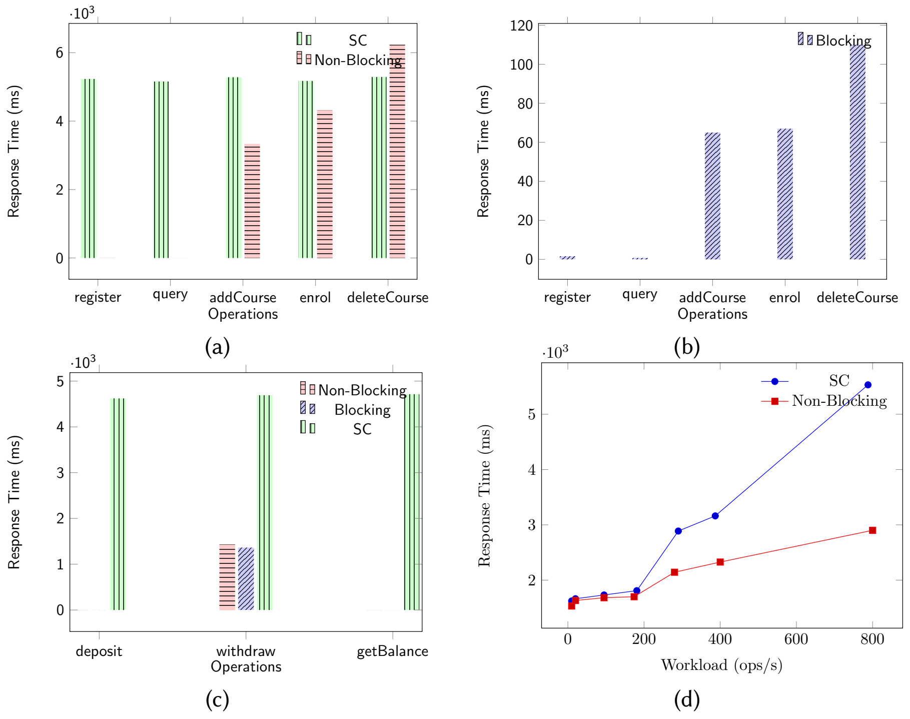


Fig. 12. (a) Response time for Courseware with the Non-Blocking and SC protocols. (b) Response time for Courseware with the Blocking protocol. (c) Response time for BankAccount. (d) The effect of workload on response time for Courseware.

automatically maps a contract to the weakest consistency notion that satisfies the contract. However, these contracts are lower-level than integrity invariants and translating invariants to contracts is non-trivial. Inspired by weak memory models, a similar work [Bernardi and Gotsman 2016; Cerone et al. 2015] presented a framework for specification of weak consistency models that have atomic visibility and defined dynamic and static checks for serializability of applications that choose to use weak consistency. Later, [Brutschy et al. 2017] defined a generalization of conflict serializability to be used together with weak consistency notions. It presented a dynamic checker to determine whether an execution of an application that uses weak consistency is serializable. In contrast, our approach requires applications to specify only higher-level integrity properties and automatically finds the coordination needed to preserve them.

Indigo [Balegas et al. 2015a,b] lets the user introduce application-specific predicates and define invariants and method post-conditions in terms of those predicates. It identifies operations that violate the invariant if executed concurrently and either prevents or repairs their concurrent execution. For the former, it applies reservation techniques to enhance coordination efficiency and for the latter, it provides a library of restoring operations. In contrast, well-coordination does not require user-defined predicates and annotations. In addition, the well-coordination conditions are formally defined and their sufficiency is proved. Further, well-coordination guarantees convergence in addition to invariant preservation. Besides, the implementation of Indigo is dependent on causal consistency of a lower-level store while we defined and implemented standalone protocols.

CISE [Gotsman et al. 2016; Najafzadeh et al. 2016] lets the user specify the invariant of the object, associate each method with tags and define conflicts between tags. It presents a rely-guarantee style proof technique for invariant preservation. The proof technique allows conditions to be associated with tags and requires that each operation guarantees the conditions of its tags relying on the conditions of non-conflicting tags. In contrast to well-coordination, the proof rule is fundamentally dependent on causal consistency and hence causal consistency is the weakest possible notion in the model. Further, our approach does not require the user to provide the conflict relation and automatically calculates it. In addition, we present protocols that provide the required coordination.

For database transactions, [Lu et al. 2004] presented correctness conditions of different isolation levels and an algorithm to find the lowest isolation level for transactions of an application to be semantically correct. Later, [Fekete 2005; Fekete et al. 2005] presented an algorithm to determine whether executions of a transaction are serializable under snapshot isolation. Recently, Alone-Together [Kaki et al. 2017] presented a program logic that enables compositional verification of invariant preservation for weakly isolated transactions. These works consider isolation levels on shared memory [Anderson et al. 2009; Hoffmann et al. 2013; Jones 1983; Lahav and Vafeiadis 2015; Nardelli et al. 2009; OHearn 2007; Owicki and Gries 1976] databases. For instance, in the AloneTogether model, all updates of a transaction become visible to all threads in an indivisible step. In contrast, we consider weak consistency for replicated state.

IPA [Holt et al. 2016] presents a type system ensuring that values from weakly consistent operations cannot flow into strongly consistency operations without explicit user endorsement. IPA stores adapt consistency to system load within the user-specified bound. Similarly, MixT [Milano and Myers 2018] is a language that allows transactions to access different stores with varying consistency guarantees and applies information flow analysis to prevent less-consistent data from influencing more-consistent data. In contrast to our approach that infers the required synchronization and dependency, IPA and MixT require the user to explicitly associate consistency conditions with objects and stores. Further, they are concerned with consistency flow rather than integrity preservation.

Epsilon-serializability [Ramamritham and Pu 1995] and TACT [Yu and Vahdat 2000] reduce coordination by bounding the staleness of replicated state. PBS [Bailis et al. 2012b] reduces coordination to a partial rather than a complete quorum and statistically bounds staleness. In contrast to bounding staleness, we focus on preserving invariants efficiently. Rationing [Kraska et al. 2009] and Pileus [Terry et al. 2013] dynamically adjust consistency based on temporal load statistics and Correctables [Guerraoui et al. 2016] incrementally makes the result more consistent to enhance responsiveness. In contrast, we presented a static approach to avoid coordination.

We note that our commutativity definitions are similar to Lipton's [Lipton 1975] moverness in nature. However, they are defined for replicated rather than shared state. In addition, they are weaker conditions. In particular, $\mathcal{P}$-commutativity does not require the same final state and return value after the move as far as the guard and the invariant continue to hold.

10 CONCLUSION

We presented an analysis and protocol co-design for automatic synthesis of correct and efficient replicated objects. We presented well-coordination, a novel sufficient condition for integrity and convergence of replicated objects that requires synchronization of conflicting and precedence of dependent methods. We presented novel parametric protocols that implement these requirements. We automatically identified conflicting and dependent methods for a suite of use-cases. We used this information to reduce the coordination avoidance problem to classical graph optimization problems and instantiated the protocols to synthesize replicated objects. The experimental results show that the synthesized objects significantly improve responsiveness.

REFERENCES

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

Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. 1995. Causal memory: Definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995).

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 (Oct. 2017), 31 pages. https://doi.org/10.1145/3110214

Zachary R. Anderson, David Gay, and Mayur Naik. 2009. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009.* 98–109. https://doi.org/10.1145/1542476.1542488

Appendix. 2018. Appendix. https://www.cs.ucr.edu/~lesani/companion/popl19/Appendix.pdf. (2018).

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

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*. ACM, 1327–1342.

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2012a. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 22.

Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. 2012b. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment* 5, 8 (2012), 776–787.

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. 2015a. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 6, 16 pages. https://doi.org/10.1145/2741948.2741972

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

Kshitij Bansal, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. 2016. A new decision procedure for finite sets and cardinality constraints in SMT. In *International Joint Conference on Automated Reasoning*. Springer, 82–98.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).

Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. 2006. PRACTI Replication. In *Proc. NSDI*.

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

Kenneth P. Birman. 1985. Replication and Fault-Tolerance in the ISIS System. In *Proc. SOSP*.

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

Coen Bron and Joep Kerbosch. 1973. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM* 16, 9 (Sept. 1973), 575–577. https://doi.org/10.1145/362342.362367

Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 458–472. https://doi.org/10.1145/3009837.3009895

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

Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer Publishing Company, Incorporated.

Domenico Cantone, Eugenio Omodeo, and Alberto Policriti. 2013. *Set theory for computing: from decision procedures to declarative programming with sets.* Springer Science & Business Media.

Domenico Cantone and Calogero G Zarba. 2000. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In *Automated Deduction in Classical and Non-Classical Logics*. Springer, 126–136.

Nuno Carvalho and et. al. 2011. APPIA Framework. http://appia.di.fc.ul.pt/wiki/index.php?title=Main_Page. (2011). Accessed: 2018-06-23.

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

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

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2 (2008), 12.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*.

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

Alan Fekete. 2005. Allocating Isolation Levels to Transactions. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '05)*. ACM, New York, NY, USA, 206–215. https://doi.org/10.1145/1065167.1065193

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.* 30, 2 (June 2005), 492–528. https://doi.org/10.1145/1071610.1071615

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

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.

Seth Gilbert and Nancy A. Lynch. 2012. Perspectives on the CAP Theorem. *IEEE Computer* 45, 2 (2012), 30–36.

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 (POPL '16)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/2837614.2837625

Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects.. In *OSDI*. 169–184.

Jan Hoffmann, Michael Marmar, and Zhong Shao. 2013. Quantitative reasoning for proving lock-freedom. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 124–133.

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 (SoCC '16)*. ACM, New York, NY, USA, 279–293. https://doi.org/10.1145/2987550.2987559

Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association.

Cliff B. Jones. 1983. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (1983), 596–619.

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2017. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proc. ACM Program. Lang.* 2, POPL, Article 27 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158115

Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay Only when It Matters. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 253–264. https://doi.org/10.14778/1687627.1687657

Viktor Kuncak and Martin Rinard. 2007. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *International Conference on Automated Deduction*. Springer, 215–230.

Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst.* 10, 4 (1992), 32.

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries reasoning for weak memory models. In *International Colloquium on Automata, Languages, and Programming*. Springer, 311–323.

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

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

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 (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292. http://dl.acm.org/citation.cfm?id=2643634.2643664

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.

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 (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 265–278. http://dl.acm.org/citation.cfm?id=2387880.2387906

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721. https://doi.org/10.1145/361227.361234

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 (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 503–517. http://dl.acm.org/citation.cfm?id=2616448.2616495

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proc. SOSP*.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proc. NSDI*.

Shiyong Lu, Arthur Bernstein, and Philip Lewis. 2004. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering* 16, 9 (2004), 1070–1081.

P. Madhusudan and P.S. Thiagarajan. 2001. Distributed Controller Synthesis for Local Specifications. In *Automata, Languages and Programming*, Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 396–407.

Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2017. Relational Constraint Solving in SMT. In *International Conference on Automated Deduction*. Springer, 148–165.

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

Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. 2016. The CISE Tool: Proving Weakly-consistent Applications Correct. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC '16)*. ACM, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/2911151.2911160

Francesco Zappa Nardelli, Peter Sewell, Jaroslav Sevcik, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. 2009. Relaxed memory models must be rigorous. In *Exploiting Concurrency Efficiently and Correctly Workshop*.

Peter W. OHearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

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 (PODC '88)*. ACM, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.62549

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 (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

Susan Owicki and David Gries. 1976. An axiomatic proof technique for parallel programs I. *Acta Informatica* 6, 4 (1976), 319–340.

Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. SOSP*.

Krithi Ramamritham and Calton Pu. 1995. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering* 7, 6 (1995), 997–1007.

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 (SIGMOD '15)*. ACM, New York, NY, USA, 1311–1326. https://doi.org/10.1145/2723372.2723720

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.

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 (PLDI '15)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

Philippe Suter, Robin Steiger, and Viktor Kuncak. 2011. Sets with cardinality constraints in satisfiability modulo theories. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 403–418.

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 309–324. https://doi.org/10.1145/2517349.2522731

Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. 1977. A New Algorithm for Generating All the Maximal Independent Sets. *SIAM J. Comput.* 6, 3 (1977), 505–517. http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=SMJCAT000006000003000505000001&idtype=cvips&gifs=yes

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

Haifeng Yu and Amin Vahdat. 2000. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 21.