Hampa: Solver-aided Recency-Aware Replication

Xiao Li, Farzin Houshmand, and Mohsen Lesani [⋆]

University of California, Riverside
{xli289, fhous001, lesani}@ucr.edu

Abstract. Replication is a common technique to build reliable and scalable systems. Traditional strong consistency maintains the same total order of operations across replicas. This total order is the source of multiple desirable consistency properties: integrity, convergence and recency. However, maintaining the total order has proven to inhibit availability and performance. Weaker notions exhibit responsiveness and scalability; however, they forfeit the total order and hence its favorable properties. This project revives these properties with as little coordination as possible. It presents a tool called Hampa that given a sequential object with the declaration of its integrity and recency requirements, automatically synthesizes a correct-by-construction replicated object that simultaneously guarantees the three properties. It features a relational object specification language and a syntax-directed analysis that infers optimum staleness bounds. Further, it defines coordination-avoidance conditions and the operational semantics of replicated systems that provably guarantees the three properties. It characterizes the computational power and presents a protocol for recency-aware objects. Hampa uses automatic solvers statically and embeds them in the runtime to dynamically decide the validity of coordination-avoidance conditions. The experiments show that recency-aware objects reduce coordination and response time.

1 Introduction

Replicated objects [13,32,45,12,23] are pervasively used for fault-tolerance, availability, responsiveness and scalability. They are used in diverse application areas [20,21,22,37,39,40,50,14,53] including embedded controllers, online services and game engines. However, coordinating the replicas has proven to be challenging. Strongly consistent replication, provided by consensus protocols such as Viewstamp [42], Paxos [34] and Raft [44], guarantees the same total order of operations across replicas. The total order simultaneously provides a hoard of favorable properties: integrity, convergence and recency. Replicas converge to the same state as the result of the same sequence of operations. Further, a propagated operation executes in the same state as the originating replica. Therefore, if an operation preserves the integrity properties [8] at the originating replica, it will certainly preserve them in the other replicas as well. In addition, the lock-step execution keeps the replicas recent: an operations executes in all replicas before the next. Thus, replicas can be stale by at most one operation.

[⋆] This project was supported by the NSF grant #1942711.

However, strong consistency may not be available and responsive during network failures or offline use. Further, its scalability is limited. The trade-off between strong consistency of replicated objects, and their availability and responsiveness is a famous dilemma [26,27,28,1,3]. Therefore, system designers opted for weaker notions of consistency such as eventual [52,48,15,17,24,19,25,4] and causal [33,2,13] consistency that can provide availability, responsiveness and scalability but lose the same total order of operations. Several projects [51,49,16] provide programming interfaces for weak consistency notions. Unfortunately, the large collection of subtle weak consistency notions is unintuitive to users. If the chosen notion is too weak, it can affect correctness, and if it is too strong, it may degrade scalability.

Therefore, researchers have recently provided high-level abstractions to shield the user from low-level complexities of weak consistency. These projects seem to be the steps towards reviving the same three pillars of consistency, i.e. integrity, convergence and recency, with as little coordination [7,47,35] as possible. CRDTs [48] revived convergence. If an object satisfies a few algebraic properties, its replication can enjoy convergence even on top of eventual consistency. However, the replicas can experience states that violate the integrity properties. Therefore, follow-up projects revived the integrity property. CISE [29] and Soteria [41] present proof techniques to verify the integrity properties of a replicated object. Sieve [36], Indigo [10] and Hamsaz [30] translate the given high-level integrity properties to hybrid models. However, they are oblivious to state recency. The operations are eventually delivered to all replicas, however, they may be arbitrarily delayed. Some updates may be delivered too late and expose the clients to stale data. On the other hand, at the expense of more communication, some updates may be immediately sent and delivered. However, applications may prefer to obtain more scalability and energy efficiency in return for bounded staleness. In fact, many applications such as ticketing, distributed sensors and network accounting can work with fairly recent data. Previous work such as TACT [55], TRAPP [43], FRACT [59], and PBS [9] considered staleness but did not address integrity and communication minimization. Further, they did not provide automatic analysis, decision and synthesis. In addition to convergence and integrity, this project, HAMPA, revives recency. Given a sequential object with the declaration of its integrity properties and recency requirements for its methods, it automatically synthesizes a correct-by-construction replicated object that guarantees integrity, convergence and recency while avoiding unnecessary coordination.

To capture object specifications from the user, we present a relational language and its denotational semantics. The language provides a complete set of relational operators to define the object methods and integrity properties, and allows the user to declare recency requirements for the return value of each method. Given a principled object specification, we present a syntax-directed analysis that infers optimum staleness bounds for each element of the state.

We present the conditions required to simultaneously preserve the three properties: convergence, integrity and recency. These conditions are used to define a novel operational semantics of replicated objects that provably preserve convergence, integrity and the inferred staleness bound. We observe that recency-awareness not only guarantees a limit on the staleness, but also allows buffering of calls and reduces the coordination required to preserve integrity.

We characterize the computational power of recency-aware replicated objects. We show that recency-aware objects have the same power as the perfect failure detector. We present a novel protocol for recency-aware replicated objects that implements the semantics. We use off-the-shelve SMT solvers both statically and embed them at runtime to decide the validity of coordination-avoidance conditions. We present a tool called HAMPA that given an object definition, analyzes the object and instantiates the protocol to synthesize replicated objects. Our experiments with the synthesized objects show that the staleness bound has an inverse relationship with the coordination and response time.

In summary, this paper presents the following contributions: (1) A relational object specification language that captures integrity and recency declarations, and its denotational semantics (§ 2). (2) The coordination conditions and the operational semantics of replicated systems that simultaneously preserve convergence, integrity and recency (§ 3 and § 4). (3) A syntax-directed analysis that infers optimum staleness bounds for each element of the state (§ 5). (4) The characterization of the computational power and a protocol for recency-aware replicated objects, (§ 6). (5) The HAMPA replicated object synthesis tool and its experimental results (§ 7). All the proofs are available in the appendix [5].

2 Recency-aware Relational Object Language

Language. Fig. 1 shows our core relational language for object specification. 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 $\mathcal{M}$. The state can be a tuple of natural number Nat and relation Rel types. The invariant $\mathcal{I}$ is a boolean function on the state. A method m is a function from the parameter x and the pre-state $\langle x_1, .., x_n \rangle$ to a record of $\langle e_g, e_u, e_r \rangle$. The guard e_g is a boolean expression that captures the semantic preconditions of m such as conditions on the arguments. The expressions e_u and e_r are for the post-state and the return value. We use guard, update and retv as functions that extract elements of this record. For each method, the user declares an integer as the staleness bound ϵ for its return value. A method call c is a method applied to its argument i.e. it is a function from the current state to a record of $\langle e_g, e_u, e_r \rangle$.

An expression e is either a value v (that can be either a number n or a relation R), a variable denoted by x, an application of the operators $\{+, -, =, <, \&, !\}$ to operand expressions where & is the conjunction and and ! is the negation operator, a selection $\sigma_{\lambda \langle \overline{x} \rangle . e}(e')$ that binds the attributes of each element of the relation e' to the variables $\overline{x}$ and returns the elements that satisfy the condition e, a projection $\Pi_{\lambda \langle \overline{x} \rangle . \langle \overline{e} \rangle}(e')$ that for each element of the relation e', binds its attributes to the variables $\overline{x}$ and calculates a tuple of elements $\langle \overline{e} \rangle$ and returns the set of resulting tuples, a union $e \cup e'$ that results in a relation with elements of both of the relations e and e', a difference $e \setminus e'$ that results in a relation with the elements in the relation e that are not in the relation e', and the Cartesian product $e \times e'$ that results in a relation with pair elements where the first and second elements are in the relations e and e' respectively. The language supports a complete set of relational operators: any relational algebra expression can be expressed by a combination of them. Selection (σ), projection (π), union ($\cup$), difference ($\setminus$), product ($\times$) and renaming (ρ) are a complete set of operators. We note that since the language uses functions with argument names, a renaming operator is unnecessary. The update and join operations are defined as a syntactic sugar. The update operation $\mathcal{U}_{\lambda\langle\overline{x}\rangle.\,\langle e,\langle\overline{e'}\rangle\rangle} e''$ returns a relation that updates each element of e'' that satisfies the condition e to the tuple $\langle\overline{e'}\rangle$. The join $e_1 \bowtie_{\lambda\langle\overline{x_1},\overline{x_2}\rangle.\,e} e_2$ results in pairs of elements of e_1 and e_2 that satisfy the condition e.

$$
\begin{array}{rll}
o := & \langle \Sigma, \mathcal{I}, \mathcal{M} \rangle & \text{Object} \\
\Sigma := & \langle T, .., T \rangle & \text{State} \\
T := & \mathsf{Nat} \ \mid \ \mathsf{Rel} & \\
\mathcal{I} & & \text{Invariant} \\
\mathcal{M} := & \overline{me} & \text{Methods} \\
me := & \mathsf{def}\ \epsilon\ m(x)(\langle x_1, .., x_n \rangle) & \text{Method} \\
 & \quad \langle e_g, e_u, e_r \rangle & \\
e := & & \text{Expression} \\
 & v & \text{Value} \\
 \mid & x & \text{Variable} \\
 \mid & e + e \ \mid \ e - e & \text{Math} \\
 \mid & e = e \ \mid \ e < e & \\
 \mid & e \,\&\, e \ \mid \ !e & \\
 \mid & \sigma_{\lambda\langle\overline{x}\rangle.e}(e) & \text{Selection} \\
 \mid & \Pi_{\lambda\langle\overline{x}\rangle.\langle\overline{e}\rangle}(e') & \text{Projection} \\
 \mid & e \cup e & \text{Union} \\
 \mid & e \setminus e & \text{Difference} \\
 \mid & e \times e & \text{Product} \\
v := & n \ \mid \ \mathsf{true} \ \mid \ \mathsf{false} \ \mid \ R & \text{Value}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{U}_{\lambda\langle\overline{x}\rangle.\,\langle e,\langle\overline{e'}\rangle\rangle} e'' := & \text{Update} \\
\quad \Pi_{\lambda\langle\overline{x}\rangle.\,\langle\overline{e'}\rangle}\, \sigma_{\lambda\langle\overline{x}\rangle.\,e}\, e'' \ \cup & \\
\quad \sigma_{\lambda\langle\overline{x}\rangle.\,\neg e}\, e'' & \\
e_1 \bowtie_{\lambda\langle\overline{x_1},\overline{x_2}\rangle.\,e} e_2 := & \text{Join} \\
\quad \sigma_{\lambda\langle\overline{x_1},\overline{x_2}\rangle.\,e}\,(e_1 \times e_2) &
\end{array}
$$

$$
[\![\, v \,]\!] = v \qquad [\![\, x \,]\!] = \bot
$$

$$
[\![\, e \oplus e' \,]\!] = [\![\, e \,]\!] \oplus [\![\, e' \,]\!] \qquad [\![\, !\, e \,]\!] = !\ [\![\, e \,]\!]
$$

$$
\begin{array}{c}
[\![\, \sigma_{\lambda\overline{x}.e'}(e) \,]\!] = \\
\{\, t \mid t \in [\![\, e \,]\!] \ \wedge \ [\![\, e'[\overline{x} \mapsto t] \,]\!] = \mathsf{true} \,\}
\end{array}
$$

$$
\begin{array}{c}
[\![\, \Pi_{\lambda\langle\overline{x}\rangle.\langle\overline{e}\rangle}(e') \,]\!] = \\
\{\, \langle \overline{[\![\, e[\overline{x} \mapsto t] \,]\!]} \rangle \mid t \in [\![\, e' \,]\!] \rangle \,\}
\end{array}
$$

$$
\begin{array}{c}
[\![\, e \cup e' \,]\!] = [\![\, e \,]\!] \cup [\![\, e' \,]\!] \\
[\![\, e \setminus e' \,]\!] = [\![\, e \,]\!] \setminus [\![\, e' \,]\!] \\
[\![\, e \times e' \,]\!] = [\![\, e \,]\!] \times [\![\, e' \,]\!]
\end{array}
$$

Fig. 1: Syntax and Semantics of the Specification Language

Semantics. Fig. 1 presents a denotational semantics for expressions. The semantics for values, variables, and binary and unary operations is standard. The semantics of the selection expression $\sigma_{\lambda\langle\overline{x}\rangle.e'}(e)$ is the set of tuples t in the semantics of e such that substitution of the attributes $\overline{x}$ in e' with their corresponding values in t evaluates to true. The semantics of the projection expression $\Pi_{\lambda\langle\overline{x}\rangle.\langle\overline{e}\rangle}(e')$ is a set of tuples, one per each tuple t in the semantics of e': a tuple resulted from substituting $\overline{x}$ with t in the expressions $\overline{e}$ and evaluating them. The semantics of union, difference and product are standard from the set theory. We define the difference Δ between two values as follows: the difference between two natural numbers is the absolute value of their subtraction i.e. $\Delta(n, n') = |n - n'|$; the difference of two relations is the size of their symmetric difference i.e. $\Delta(R, R') = |R \setminus R'| + |R' \setminus R|$. We use delta δ to represents the staleness of a value that is the difference between the value and its target value. The delta for a completely recent (or exact) value is zero. For a call

```
Class MovieBooking
    $\Sigma := \mathsf{let}\ rs := \mathsf{Set}\ \mathbb{N} \times \mathbb{N}\ \mathsf{in}$    $\triangleright$ Reservation: user identifier and movie identifier
        $\mathsf{let}\ ms := \mathsf{Set}\ \mathbb{N} \times \mathbb{N}\ \mathsf{in}$    $\triangleright$ Movie: movie identifier and available space
        $\langle rs, ms \rangle$
    $\mathcal{I} := \lambda \langle rs, ms \rangle.\ \mathsf{unique}\,(ms, \lambda \langle m, a \rangle.\ m) \wedge$
                    $\mathsf{refIntegrity}\,(rs, \lambda \langle u, m \rangle.\ m, ms, \lambda \langle m, a \rangle.\ m) \wedge$
                    $\mathsf{rowIntegrity}\,(ms, \lambda \langle m, a \rangle.\ a \geq 0)$
    $\mathsf{book}(\langle u, m \rangle) := 0\ \lambda \langle rs, ms \rangle.$
        $\langle \langle u, m \rangle \notin rs, \quad \langle rs \cup \langle u, m \rangle,\ \mathcal{U}_{\lambda \langle m', a \rangle.\ \langle m'=m, \langle m, a-1 \rangle \rangle}\, ms \rangle, \quad \bot \rangle$
    $\mathsf{cancelBook}(\langle u, m \rangle) := 0\ \lambda \langle rs, ms \rangle.$
        $\langle \mathsf{True}, \quad \langle rs \setminus \langle u, m \rangle,\ \mathcal{U}_{\lambda \langle m', a \rangle.\ \langle m'=m, \langle m, a+1 \rangle \rangle}\, ms \rangle, \quad \bot \rangle$
    $\mathsf{offScreen}(m) := 0\ \lambda \langle rs, ms \rangle.$
        $\langle \mathsf{True}, \quad \langle rs,\ ms \setminus \sigma_{\lambda \langle m', a \rangle.\ m'=m}\, ms \rangle, \quad \bot \rangle$
    $\mathsf{specialReserve}(\langle m, n \rangle) := 0\ \lambda \langle rs, ms \rangle.$
        $\langle n > 0, \quad \langle rs,\ \mathcal{U}_{\lambda \langle m', a \rangle.\ \langle m'=m, \langle m, a-n \rangle \rangle}\, ms \rangle, \quad \bot \rangle$
    $\mathsf{increaseSpace}(\langle m, n \rangle) := 0\ \lambda \langle rs, ms \rangle.$
        $\langle n > 0, \quad \langle rs,\ \mathcal{U}_{\lambda \langle m', a \rangle.\ \langle m'=m, \langle m, a+n \rangle \rangle}\, ms \rangle, \quad \bot \rangle$
    $\mathsf{querySpace}(m) := \epsilon_1\ \lambda \langle rs, ms \rangle.$
        $\langle \mathsf{True}, \quad \langle rs, ms \rangle, \quad \mathit{\Pi}_{\lambda \langle m', a \rangle.\ \langle a \rangle}\, (\sigma_{\lambda \langle m', a \rangle.\ m'=m}\, ms) \rangle$
    $\mathsf{queryReservations}(u) := \epsilon_2\ \lambda \langle rs, ms \rangle.$
        $\langle \mathsf{True}, \quad \langle rs, ms \rangle, \quad \mathit{\Pi}_{\lambda \langle u', m \rangle.\ \langle m \rangle}\, (\sigma_{\lambda \langle u', m \rangle.\ u'=u}\, rs) \rangle$
    $\mathsf{querySpaces}(u) := \epsilon_3\ \lambda\, \langle rs, ms \rangle.$
        $\langle \mathsf{True}, \quad \langle rs, ms \rangle, \quad \mathit{\Pi}_{\lambda \langle u, m, m', a \rangle\ \langle m, a \rangle}\, (rs \bowtie_{\lambda \langle u, m \rangle, \langle m', a \rangle.\ m=m'}\ ms) \rangle$
```

Fig. 2: Movie Booking Use-case

c, the weight $\mathsf{weight}(c)$ is a bound on the difference that the execution of c can make on the state of the object. In other words, for every call c, we have $\forall \sigma$. Let $\langle \_, \sigma', \_ \rangle := c(\sigma)$ in $\mathit{\Delta}(\sigma', \sigma) < \mathsf{weight}(c)$.

Running Use-case. Fig. 2 shows the movie booking use-case. The state of the object is the two relations reservation rs and movie ms. The reservation relation rs stores the movies that the users have booked; it is the pairs of users u and movies m. The movie relation ms stores the number of available spaces for each movie; it is the pairs of movies m and spaces a. The integrity property $\mathcal{I}$ is a conjunction of three conditions: (1) The movie in ms should be unique. (2) The referential integrity requires that every movie in rs exists in ms. (3) The number of available spaces for every movie should be non-negative. The object provides five update methods and three query methods. Given a user u and a movie m, the method book adds the pair to rs and decrements the available spaces for m in ms. Similarly, the method $\mathsf{cancelBook}$ removes a reservation and increments available spaces. Given a movie m, the method $\mathsf{offScreen}$ removes the corresponding tuple from ms. Given a movie m and a number n, the method $\mathsf{specialReserve}$ subtracts n from the available spaces for m in ms. The dual method $\mathsf{increaseSpace}$ adds n to the spaces for m. Given a movie m, the method $\mathsf{querySpace}$ returns the number of available spaces for m. The method $\mathsf{queryReservations}$ returns the set of movies that the given user has booked. Given a user u, the method querySpaces returns the pairs of movies and their available spaces for the movies that u has booked. The staleness bound for the update methods is specified as 0. The returned none constant $\perp$ is always exact. The bound values ϵ_1, ϵ_2 and ϵ_3 of the query methods represent the number of tuples that are different between the current state and the pending stable state of the result relation.

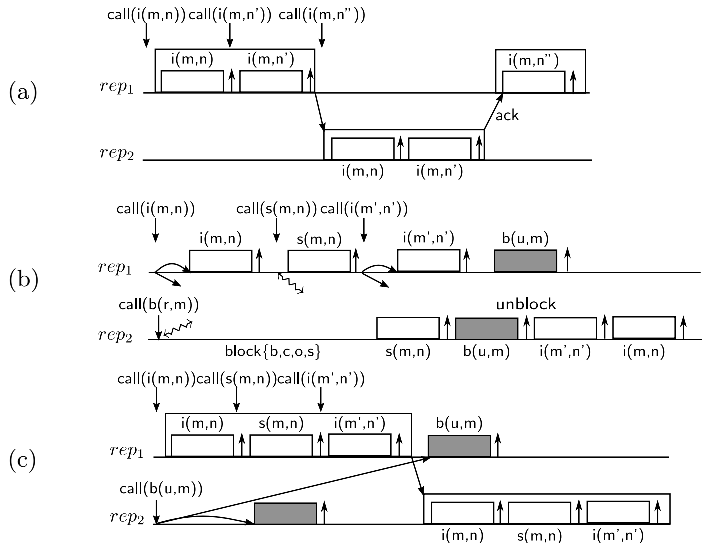

Fig. 3: (a) Buffering and coordination. Example execution (b) without and (c) with recency. ↓: request, ↑: indication, ⇝: synchronization

To reduce communication, certain calls can be executed locally and buffered, and the buffer can be communicated to other replicas later. As an example, in Fig. 3.(a), the first two calls to the method increaseSpace do not exceed the staleness bound for *ms* and can be buffered. However, the third call exceeds the bound and cannot be added to the buffer. Therefore, the buffer is flushed to other replicas and the third call is blocked until an acknowledgement for the delivery of the buffer is received. All the calls of the buffer can be sent in a single message and the acknowledgement for them can be sent in a single message as well.

Let us now consider the interaction of buffering with coordination. We will see that buffering (staleness) interestingly reduces the coordination required for the conflicts. (We will define conflicting calls that should be synchronized later in § 3.) Fig. 3.(b) and (c) show the same execution without and with buffering respectively. In Fig. 3.(b), the first replica rep_1 executes the sequence of calls increaseSpace, specialReserve and increaseSpace. The method increaseSpace does not conflict with any other method; therefore, calls to it are simply broadcast. The method specialReserve conflicts with itself and the method book; therefore, the call to it goes through synchronization. The second replica rep_2 calls book that conflicts with four other methods. Hence, it should synchronize. (The synchronization reaches to other replicas, blocks calling the four methods, and propagates previous calls to those methods.) In this example, the conflicting specialReserve call in rep_1 should be propagated to rep_2 before the book call can be executed.

In Fig. 3.(c), the recency bound allows the three calls of rep_1 to be buffered. Replicas use SMT solvers at runtime to check the validity of three properties for the buffers: all-$\mathcal{S}$-commutativity, invariant-sufficiency and let-$\mathcal{P}$-R-commutativity that we will formally define in § 3. In this example, the buffer is invariant-sufficient if the number of spaces that the call specialReserve decrements is less than the number that the increaseSpace calls increment. Therefore, the buffer can be sent to other replicas without any additional synchronization; the invariant in the pre-state is sufficient for the invariant in its post-state. We note that the call specialReserve that previously went through synchronization does not need any synchronization inside the buffer. Further, the let-$\mathcal{P}$-R-commutativity property of the buffer guarantees that the book call will preserve the integrity after the buffer. Thus, the synchronization of the book call that previously waited for the specialReserve call does not need to wait anymore.

3 Coordination Conditions

In this section, we present the coordination conditions for replicated objects that preserve the three properties: convergence, integrity and recency. The state of the given sequential object is replicated across replicas. Clients can request method calls at every replica, and replicas coordinate the calls. *Convergence* is the safety property that when all pending updates are processed, the replicas converge to the same state. *Integrity* is the safety property that every method call is executed only on a state where the guard of the method and the invariant are satisfied. *Recency* is the safety property that bounds the difference between the state of a replica and its impending state after the pending calls are applied.

The state of each replica is initialized to the same state σ_0 that satisfies the invariant $\mathcal{I}$. The replica that accepts the request for a call from the user is called the originating replica of the call. We uniquely identify requests by identifiers r. We use the two maps call and orig that map request identifiers to the method call and originating replica respectively. The execution history of a replica is modeled as a permutation of a set of request identifiers. An execution x of a set of requests R is a bijective from positions $[0..|R|-1]$ to R. We denote the range of x as $R(\mathsf{x})$. An execution x of R defines the total order $\prec_{\mathsf{x}}$ on R: A request r precedes another request r' in an execution x written as $r \prec_{\mathsf{x}} r'$ iff $\mathsf{x}^{-1}(r) < \mathsf{x}^{-1}(r')$. A replicated execution xs is a function from replicas $\mathcal{N}$ to executions. The post-state of each call at a replica is the result of applying the call to its pre-state.

We first revisit the coordination conditions for convergence and integrity [30], and then present coordination conditions for recency and their impact on the prior conditions.

Convergence. A replicated execution is convergent if the state of the replicas is the same after all the calls are propagated. Out of order delivery of method calls at different replicas can lead to divergence of their states. Method calls such as special reservation specialReserve and increasing space increaseSpace result in the same state if their order of execution is swapped. However, the resulting state of the two method calls book and cancelBook is dependent on their execution order. Therefore, they should synchronize.

Definition 1 (State-Commutativity and State-Conflict). *Two method calls* c_1 *and* c_2 $\mathcal{S}$*-commute, written as* $c_1 \leftrightarrows_{\mathcal{S}} c_2$ *iff for every state* σ, $\mathsf{update}(c_2)(\mathsf{update}(c_1)(\sigma)) = \mathsf{update}(c_1)(\mathsf{update}(c_2)(\sigma))$*. Otherwise, they* $\mathcal{S}$*-conflict, written as* $c_1 \bowtie_{\mathcal{S}} c_2$*.*

Integrity. The body of each method relies on the invariant in the pre-state. Further, methods have explicit guards that declare their pre-conditions. We say that a method call enjoys integrity at a state if the invariant and the guard of the method hold in that state.

Definition 2 (Integrity). *A method call c enjoys integrity in a state* σ*, written as* $\mathsf{integrity}(\sigma, c)$*, iff* $\mathsf{guard}(c)(\sigma)$ *and* $\mathcal{I}(\sigma)$*.*

Method calls should be executed only in states that they have integrity in. The integrity condition is simply lifted to executions and replicated executions: An execution enjoys integrity iff every request in it enjoys integrity.

Definition 3 (Permissibility). *A method call c is permissible in a state* σ*, written as* $\mathcal{P}(\sigma, c)$*, iff* $\mathsf{guard}(c)(\sigma)$ *and* $\mathcal{I}(\mathsf{update}(c)(\sigma))$*.*

In contrast to integrity that requires the invariant to hold in the pre-state, permissibility requires it to hold in the post-state. The post-state of a call is the pre-state of the next call in a replica. Further, the initial state is assumed to satisfy the invariant. Therefore, if every call is permissible in its pre-state, then every call enjoys integrity. By induction, permissibility leads to integrity.

To execute a method call, we check that it is permissible at its originating replica. Thus, we say that each method call is *locally permissible*. Otherwise, the call is aborted or delayed. Still, if the call is simply broadcast, it is not necessarily permissible when it arrives at other replicas. Some calls need coordination.

Conflict. There are calls such as increaseSpace that are always permissible as far as they are applied to a state that satisfies the invariant. Increasing the space cannot result in a missing or duplicate movie or a negative number for available spaces. Thus, if it is broadcast and executed on another replica, it is sufficient that the pre-state satisfies the invariant to preserve it in the post-state.

Definition 4 (Invariant-Sufficient). *A call c is invariant-sufficient iff for every state* σ*, if* $\mathcal{I}(\sigma)$ *then* $\mathcal{P}(\sigma, c)$*.*

However, not all calls are invariant-sufficient. For example, a book call may be permissible in a replica but may become impermissible in another when it is executed after an already executed offScreen call for the same movie. These two calls should synchronize to preserve integrity. Nonetheless, some pairs of calls such as offScreen and specialReserve do not affect each other's permissibility. (In the running example, specialReserve has no guards. After an offScreen call, it remains permissible as it doesn't find the movie and leaves the relation unchanged).

Definition 5 (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 $\mathcal{P}(\mathsf{update}(c_2)(\sigma), c_1)$.*

If a call c_1 is invariant-sufficient or $\mathcal{P}$-R-commutes another call c_2, then the call c_1 will stay permissible when it is propagated and applied to another replica even if c_2 is executed before it in that replica.

Definition 6 (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.*

The call offScreen $\mathcal{P}$-concurs with the call specialReserve; however, the call book $\mathcal{P}$-conflicts with the call offScreen.

We say that two calls *concur* iff they both $\mathcal{S}$-commute and $\mathcal{P}$-concur with each other. Otherwise, we say they *conflict* and need synchronization.

Definition 7 (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 $c_1 \bowtie c_2$.*

Dependency. As we saw above, invariant-sufficient method calls can 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. For example, taking the movie off-screen offScreen is dependent on cancelling the last booking cancelBook. If offScreen is moved left before cancelBook, it can become impermissible. Nonetheless, taking a movie off-screen offScreen is independent of the previous special reservations specialReserve.

Definition 8 (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 σ, if $\mathcal{P}(\mathsf{update}(c_1)(\sigma), c_2)$ 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 9 (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 $c_2 \not\perp\!\!\!\perp c_1$.*

If c_1 is executed before c_2 in the originating replica of c_2 and c_2 is dependent on c_1, then c_2 should be applied to other replicas only if c_1 is already applied.

Recency. Calls executed at a replica may be delayed in the network before they are executed in other replicas. Further, they may be buffered at the originating replica to reduce communication. The *pending* calls for a replica are the calls that have executed in other replicas but not at that replica yet. The staleness of a replica is the difference of its current state and its state after applying its pending calls. Given a bound ϵ, a replica is sufficiently recent if its staleness is less than ϵ. The calls that have originated in the current replica n but have not been received yet by another replica n' make the state of n'

stale. To bound the staleness of n' by ϵ, the staleness imposed to n' by the calls originated by each of the other $|\mathcal{N}|-1$ replicas should be bounded by $\epsilon/(|\mathcal{N}|-1)$. The difference that these calls can make is bounded by the sum of their weights (defined in § 2). The staleness bound can be evenly divided between the replicas. However, in general it can be distributed unevenly and even dynamically. In particular, replicas that tend to issue updates more often can get a larger share.

Given a recency bound, a buffering quota can be calculated for each replica and the recency bound can be preserved when calls are buffered. Buffering calls can reduce communication; however, it can affect the convergence and integrity properties. To preserve these properties a buffer should have three properties: all-state-commutativity, invariant-sufficiency and let-$\mathcal{P}$-R-commutativity. We consider each condition in turn.

Definition 10 (All-State-Commutative). *A call is all-$\mathcal{S}$-commutative if it is $\mathcal{S}$-commutative with respect to every call.*

The calls of the buffer are executed locally and are not synchronized with other replicas. Therefore, if the buffer is not all-$\mathcal{S}$-commutative, concurrent execution of $\mathcal{S}$-conflicting calls in other replicas can lead to divergence. Similarly, if the buffer is not invariant-sufficient, concurrent execution of $\mathcal{P}$-conflicting calls in other replicas can lead to impermissiblility of the buffer when it is propagated and executed in other replicas. The buffer in Fig. 3.(c) is all-$\mathcal{S}$-commutative: it includes increaseSpace and specialReserve calls that result in increasing or decreasing the space for movies; the result is $\mathcal{S}$-commutative with respect to all method calls. Further, it is invariant-sufficient if the net result of its calls is a non-negative addition to the space of each movie. For example, if the increaseSpace calls add s spaces and the specialReserve calls subtract s' spaces from the same movie where $s' \leq s$, then the net effect is adding spaces and the buffer is invariant-sufficient.

Definition 11 (Let-Permissible-Right-Commutative). *A call is let-$\mathcal{P}$-R-commutative if every call $\mathcal{P}$-R-commutes with it.*

Calls in other replicas are checked to be permissible with no knowledge of the buffered calls in the current replica. Let-$\mathcal{P}$-R-commutativity of the buffer of the current replica guarantees that the calls in other replicas will continue to be permissible once they are propagated and executed after the buffer in the current replica. The buffer in Fig. 3.(c) is let-$\mathcal{P}$-R-commutative; it may only increase the number of spaces that cannot make any call impermissible.

4 Replicated System Semantics

In this section, we define the operational semantics of replicated objects where (1) the integrity property $\mathcal{I}$ on the state of each replica is always preserved, (2) replicas converge to the same state once all the calls are propagated, and (3) the staleness of each replica is always bounded by ϵ. The semantics declares the conditions for execution and propagation of method calls on the replicated object to guarantee the three properties. In particular, it represents the conditions for local buffering of method calls to avoid communication while preserving the recency of the other replicas. In § 5, we will see a static analysis that infers staleness bounds for the state. In this section, the semantics preserves the inferred staleness bound ϵ for the state σ of the object. (For objects with multiple pieces of state, the staleness of each piece can be tracked separately.) The semantics strives to concisely define the conditions; we will present the protocols that implement these conditions in § 6.

As Fig. 4 shows, the global state of the replicated system is represented as a world w that is a tuple of $\langle h, t, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\rangle$. The hosts h is a mapping from replica identifiers $\mathcal{N}$ to the local state of replicas. Each call is assigned a unique request identifier r at the originating replica. The two maps call and orig keep a mapping from request identifiers to the call and the originating replica of the request respectively. The state of each replica is a statement $s \in S$, the state of the object $\sigma \in \Sigma$, and the identifier $r \in R$ of the current buffer. A statement s is either $x \leftarrow c; s'$ that is the sequence of a call c and another statement s', or the terminal statement skip. A call c is the application of a method m to an argument expression e. A call can also be the identity call id that leaves the state unchanged. (It is assumed that client statements do not make id calls.) The network t is the set of packets that are sent but not yet delivered. A packet p contains the identifier of the destination replica n and the request identifier r of the call. If a packet is transmitting a buffered call, it is decorated with an asterisk $*$. The history xs is a mapping from replica identifiers $\mathcal{N}$ to the list of request identifiers of the calls that are previously applied to that replica. The initial value of the world state is w_0 where each replica n hold its initial statement s_n, the initial state σ_0 of the object that satisfies the integrity property $\mathcal{I}$, and an empty buffer. Empty buffers are represented by mapping the buffer identifier r_n of each replica n to the identity call id.

w	$:=$	$\langle h, t, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\rangle$	World
h	$:$	$\mathcal{N} \mapsto S \times \Sigma \times R$	Hosts
n	$:$	$\mathcal{N}$	Replica nodes
$s : S$	$:=$	$x \leftarrow c;\ s \mid \mathsf{skip}$	Statement
$c : C$	$:=$	$m(e) \mid \mathsf{id}$	Call
m	$:$	$\mathcal{M}$	Method
e	$:=$	$x \mid v$	Expression
x			Variable
v			Value
σ	$:$	Σ	Object State
r	$:$	R	Request
t	$:$	$\mathsf{Set}\ P$	Transit
$p : P$	$:=$	$\langle n, r\rangle \mid \langle n, r^*\rangle$	Packet
xs	$:$	$\mathcal{N} \mapsto \mathsf{List}\ R$	History
orig	$:$	$R \mapsto \mathcal{N}$	Original node
call	$:$	$R \mapsto C$	Request call
w_0	$:=$	$\langle \overline{n \mapsto \langle s_n, \sigma_0, r_n\rangle}_{n\in\mathcal{N}}, \emptyset, \emptyset, [\overline{r_n \mapsto n}]_{n\in\mathcal{N}}, [\overline{r_n \mapsto \mathsf{id}}]_{n\in\mathcal{N}}\rangle$	Init World

Fig. 4: Operational Semantics State

Fig. 5 presents the operational semantics. The rule CALL executes a method call c at a replica n. The call c can be executed if the following conditions hold. (1) To preserve integrity, the call c should be locally permissible $\mathcal{P}(\sigma, c)$ in the current state σ. (2) To preserve convergence and integrity, any pair of conflicting calls should have the same order across the replicas, a property that we call conflict-synchronization. Thus, to execute a new request r, the rule CALL requires

Call

$$\frac{\begin{array}{c}\mathcal{P}(\sigma, c) \qquad c(\sigma) = \langle \_, \sigma', v\rangle \\ \text{fresh } r \qquad \mathsf{orig}' = \mathsf{orig}[r \mapsto n] \\ \mathsf{call}' = \mathsf{call}[r \mapsto c] \\ \mathsf{xs}' = \mathsf{xs}[n \mapsto (\mathsf{xs}(n) ::: r)] \\ \mathsf{ConflictSyncInit}_{\langle \mathsf{call}' \rangle}(\mathsf{xs}', n, r) \\ \mathsf{InBound}_{\langle \mathsf{orig}', \mathsf{call}' \rangle}(\mathsf{xs}', n) \\ t' = t \cup \{\langle n', r\rangle \mid n' \in \mathcal{N} \setminus \{n\}\}\end{array}}{\begin{array}{c}(h[n \mapsto (x \leftarrow c; s,\ \sigma,\ r')], t, \mathsf{xs}, \mathsf{orig}, \mathsf{call}) \\ \xrightarrow{n,r,c} \\ (h[n \mapsto (s[x \mapsto v],\ \sigma',\ r')], t', \mathsf{xs}', \mathsf{orig}', \mathsf{call}')\end{array}}$$

Deliver

$$\frac{\begin{array}{c}\mathsf{call}(r)(\sigma) = \langle \_, \sigma', \_\rangle \\ \mathsf{xs}' = \mathsf{xs}[n \mapsto (\mathsf{xs}(n) ::: r)] \\ \mathsf{ConflictSync}_{\langle \mathsf{call} \rangle}(\mathsf{xs}', n, r) \\ \mathsf{DepPres}_{\langle \mathsf{orig}, \mathsf{call} \rangle}(\mathsf{xs}', n, r)\end{array}}{\begin{array}{c}(h[n \mapsto (s, \sigma, r')], t \cup \{\langle n, r\rangle\}, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\}) \\ \xrightarrow{n,r,\mathsf{call}(r)} \\ (h[n \mapsto (s, \sigma', r')], t, \mathsf{xs}', \mathsf{orig}, \mathsf{call})\end{array}}$$

CallLocal

$$\frac{\begin{array}{c}\mathcal{P}(\sigma, c) \qquad c(\sigma) = \langle \_, \sigma', v\rangle \\ c' = c \cdot \mathsf{call}(r) \\ \mathsf{AllSComm}(c) \\ \mathsf{InvSuff}(c') \qquad \mathsf{LetPRComm}(c') \\ \mathsf{call}' = \mathsf{call}[r \mapsto c'] \\ \mathsf{xs}' = \begin{cases} \mathsf{xs}[n \mapsto (\mathsf{xs}(n) ::: r)] & \text{if } \mathsf{call}(r) = \mathrm{id} \\ \mathsf{xs} & \text{else}\end{cases} \\ \mathsf{InBound}_{\langle \mathsf{orig}, \mathsf{call}' \rangle}(\mathsf{xs}', n)\end{array}}{\begin{array}{c}(h[n \mapsto (x \leftarrow c; s,\ \sigma,\ r)], t, \mathsf{xs}, \mathsf{orig}, \mathsf{call}) \\ \xrightarrow{n,r,c} \\ (h[n \mapsto (s[x \mapsto v],\ \sigma',\ r)], t, \mathsf{xs}', \mathsf{orig}, \mathsf{call}')\end{array}}$$

SendBuffer

$$\frac{\begin{array}{c}\mathsf{call}(r) \neq \mathrm{id} \qquad \text{fresh } r' \\ \mathsf{orig}' = \mathsf{orig}[r' \mapsto n] \qquad \mathsf{call}' = \mathsf{call}[r' \mapsto \mathrm{id}] \\ t' = t \cup \{\langle n', r^*\rangle \mid n' \in \mathcal{N} \setminus \{n\}\}\end{array}}{\begin{array}{c}(h[n \mapsto (s, \sigma, r)], t, \mathsf{xs}, \mathsf{orig}, \mathsf{call}) \\ \longrightarrow \\ (h[n \mapsto (s, \sigma, r')], t', \mathsf{xs}, \mathsf{orig}', \mathsf{call}')\end{array}}$$

DeliverBuffer

$$\frac{\begin{array}{c}\mathsf{call}(r)(\sigma) = \langle \_, \sigma', \_\rangle \\ \mathsf{xs}' = \mathsf{xs}[n \mapsto (\mathsf{xs}(n) ::: r)]\end{array}}{\begin{array}{c}(h[n \mapsto (s, \sigma, r')], t \cup \{\langle n, r^*\rangle\}, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\}) \\ \xrightarrow{n,r,\mathsf{call}(r)} \\ (h[n \mapsto (s, \sigma', r')], t, \mathsf{xs}', \mathsf{orig}, \mathsf{call})\end{array}}$$

$$\begin{array}{rcl}
\mathrm{id} & := & \lambda\sigma.\ \langle \mathsf{True}, \sigma, \bot\rangle \\
\mathcal{P}(\sigma, c) & := & \text{Let } \langle g, \sigma', \_\rangle := c(\sigma) \text{ in } (g = \mathsf{true} \wedge \mathcal{I}(\sigma') = \mathsf{true}) \\
\mathsf{ConflictSyncInit}_{\langle \mathsf{call} \rangle}(\mathsf{xs}, n, r) & := & \forall n', r'.\ r' \in \mathsf{xs}(n') \wedge \mathsf{call}(r) \bowtie \mathsf{call}(r') \rightarrow r' \in \mathsf{xs}(n) \\
\mathsf{ConflictSync}_{\langle \mathsf{call} \rangle}(\mathsf{xs}, n, r) & := & \forall n', r'.\ r' \prec_{\mathsf{xs}(n')} r \wedge \mathsf{call}(r) \bowtie \mathsf{call}(r') \rightarrow r' \prec_{\mathsf{xs}(n)} r \\
\mathsf{DepPres}_{\langle \mathsf{orig}, \mathsf{call} \rangle}(\mathsf{xs}, n, r) & := & \forall r'.\ r' \prec_{\mathsf{xs}(\mathsf{orig}(r))} r \wedge \mathsf{call}(r) \not\bowtie \mathsf{call}(r') \rightarrow r' \in \mathsf{xs}(n) \\
\mathsf{AllSComm}(c) & := & \forall c'.\ c \leftrightarrows_{\mathcal{S}} c' \\
\mathsf{InvSuff}(c) & := & \forall \sigma.\ \mathcal{I}(\sigma) \rightarrow \mathcal{P}(\sigma, c) \\
\mathsf{LetPRComm}(c) & := & \forall c'.\ c' \rightarrow_{\mathcal{P}} c \\
\mathsf{InBound}_{\langle \mathsf{orig}, \mathsf{call} \rangle}(\mathsf{xs}, n) & := & \forall n'.\ \sum_{r \in \mathsf{xs}(n) \setminus \mathsf{xs}(n') \wedge \mathsf{orig}(r) = n} \mathsf{weight}(\mathsf{call}(r)) < \frac{\epsilon}{|\mathcal{N}|-1} \\
(c \cdot c')(\sigma) & := & \text{Let } \langle \_, \sigma', \_\rangle := c'(\sigma) \text{ in } c(\sigma')
\end{array}$$

Fig. 5: Replicated System Semantics

the condition ConflictSyncInit: any call r' that is already executed in another replica n' and conflicts with the current call r should have been already executed in the current replica n. Otherwise, once the calls r and r' are propagated and executed on the other replicas, they will have different orders in the two replicas

n and n'. (3) To preserve recency, this rule requires the condition $\mathsf{InBound}$: the difference that the pending calls from the current replica n can make to the state of every other replica n' should be bounded by $\epsilon/(|\mathcal{N}|-1)$. If the conditions above hold, a fresh identifier r is created for the call, the history xs and the maps orig and call are updated to reflect the new call, a packet is sent in the network t to every other replica, and the variable x is substituted with the returned value v of the call in the continuation statement s of the current replica.

The rule DELIVER delivers a call that has been sent to the current replica. It requires two conditions: conflict-synchronization and dependency-preservation. (1) Similar to the rule CALL, conflict-synchronization requires $\mathsf{ConflictSync}$: if a conflicting call r' is executed before the received call r in another replica n', then r' should have been already executed before r in n as well. (2) To preserve integrity, the dependencies of calls should be preserved. Thus, the dependency-preservation condition $\mathsf{DepPres}$ requires that a call r originated from a replica n' is executed in the current replica n only if the calls r' that have been executed before r in n' and r is dependent on r' should have been already executed in n.

Recency-aware replication can be applied to any object, but it can improve performance when there are method calls that can be buffered. The rule CALLLOCAL executes a call but locally buffers it. Similar to the rule CALL, it first checks the local permissibility of the call c. Since a buffered call is not immediately coordinated with calls in other replicas, it should satisfy the three properties (that saw in § 5) to make it concur with any call: (1) all-state-commutativity $\mathsf{AllSComm}$, (2) invariant-sufficiency $\mathsf{InvSuff}$, and (3) let-$\mathcal{P}$-Right-commutativity $\mathsf{LetPRComm}$. The identifier of the current buffer is r; the current call c is composed with the current buffered call $\mathsf{call}(r)$ to result in a composed call c' for the updated buffer. The composition $\cdot$ of calls simply cascades their updates to the state. The all-state-commutativity condition is stated for single calls c (that implies the same condition for the composed call c' as well). This condition is required for the call c because there might be other calls delivered between the last buffered call and the currently buffered call c. The call c should state-commute past the calls in between. Further, as explained for the rule CALL, the condition $\mathsf{InBound}$ requires that the added staleness remains within bound. If the above conditions hold, the map call is updated with the new buffer call c', and the identifier r of the buffered call is added to the history xs, if the buffer was empty and the current call c is the first buffered call.

The rule SENDBUFFER sends the buffer to every other replica and resets the buffer. Packets transmitting buffers are decorated with an asterisk. The rule DELIVERBUFFER receives a packet containing a buffer. As we saw in the rule CALLLOCAL, buffers are checked to be invariant-sufficient in the originating replica. Therefore, on receiving a packet containing a buffer, in contrast to the rule DELIVER, the rule DELIVERBUFFER does not checks the dependency-preservation$\mathsf{DepPres}$ and the conflict-synchronization $\mathsf{ConflictSync}$ conditions.

The following lemmas state the three properties of the semantics. The following lemma states that once the buffers are flushed $\mathsf{call}(r) = \mathsf{call}(r') = \mathsf{id}$ and the messages are delivered $t = \emptyset$, the replicas converge to the same state.

Lemma 1 (Convergence). *For all h, n, n', σ, σ', r and r', if $w_0 \longrightarrow^* \langle h, \emptyset, \_, \_, \_\rangle$ where $h(n) = \langle \_, \sigma, r\rangle$, $h(n') = \langle \_, \sigma', r'\rangle$ and $\mathsf{call}(r) = \mathsf{call}(r') = \mathsf{id}$ then $\sigma = \sigma'$.*

The following lemma states that every call enjoys the integrity property.

Lemma 2 (Integrity). *For all h, n, r, c, w and σ, if $w_0 \longrightarrow^* \langle h, \_, \_, \_, \_\rangle \xrightarrow{n,\_,c} w$ where $h(n) = \langle \_, \sigma, \_\rangle$ then $\mathsf{integrity}(\sigma, c)$.*

The staleness of a replica is the difference of its current state and its state after applying its pending calls from others (buffered calls and in transit calls). The following lemma states that the stateless of every replica is bounded by ϵ.

Lemma 3 (Recency). *For all h, h', n, s, σ and σ', if $w_0 \longrightarrow^* \langle h, \_, \_, \_, \_\rangle (\longrightarrow \cup \xrightarrow{n,\_,\_})^* \langle h', \_, \_, \_, \_\rangle$, $h(n) = \langle s, \sigma, \_\rangle$, and $h'(n) = \langle s, \sigma', \_\rangle$ then $\Delta(\sigma', \sigma) < \epsilon$.*

5 Staleness Bound Inference and Optimization

In § 4, we presented an operational semantics that preserves a given staleness bound for the state. The users declare the recency that they expect from the return value of each method of the object. The specified bounds for the methods can be used to infer the bounds for the elements of the state. In this section, given an object specification that includes recency declarations for the methods, we present a static analysis that infers optimum staleness bounds for each element of the state. We present a syntax-directed analysis that derives recency constraints between bound variables for the state elements. A solution to the constraints assigns a bound value to each state element such that if every state element keeps its staleness bound then the result of every method call respects the recency declaration of the method. The optimum solution maximizes the (weighted) sum of the bounds to increase buffered calls and hence decrease communication.

Fig. 6 presents the constraint inference rules for the object language that we saw in Fig. 1. A delta bound δ is either a natural number n, a delta variable dx, or addition or multiplication of two deltas. A constraint C is equality or comparison of two deltas, or conjunction of two constraints. A delta environment Γ is a mapping from variables to delta variables or values. The judgements are of the following forms: the judgement $o \triangleright C$ states the bounding constraint C for the object o, the judgement $m \triangleright C$ states the constraint C for the method m, and the judgement $\Gamma \vdash e \triangleright \delta, C$ states that under the delta environment Γ, the staleness of the expression e is bounded by δ when the constraints C are satisfied. The rule COBJ states that the constraint for an object is the conjunction of the constraints for its methods. (We assume that the state variables passed to all the methods are renamed to the same variables $\langle \sigma_1, .., \sigma_n\rangle$.) The rule CMET infers the constraints for a method by first, inferring the constraints for its return expression under a delta environment where the argument is mapped to the delta value of zero (exactly recent) and the state variables σ_i are mapped to delta variables $d\sigma_i$ to be inferred, and second, bounding the return value. The

$$\begin{array}{l} \delta := n \mid dx \mid \delta+\delta \mid \delta\times\delta \\ C := \delta=\delta \mid \delta<\delta \mid C\wedge C \end{array} \quad \text{Bound Constraints} \qquad \text{CObj}\ \frac{\overline{me \rhd C}}{\langle \Sigma, \mathcal{I}, \overline{me}\rangle \rhd \wedge\overline{C}}$$

$$\text{CMet}\ \frac{\begin{array}{c}\mathsf{free}(e_r) = \{x, \sigma_1, .., \sigma_n\} \\ [x \mapsto 0, \sigma_1 \mapsto d\sigma_1, .., \sigma_n \mapsto d\sigma_n] \vdash e_r \rhd \delta', C\end{array}}{\mathsf{def}\ \delta\ m(x)(\langle \sigma_1, .., \sigma_n\rangle)\ \langle e_g, e_u, e_r\rangle \rhd C \wedge (\delta' \leq \delta)} \qquad \text{CVal}\ \frac{}{\Gamma \vdash v \rhd 0, \emptyset} \qquad \text{CVar}\ \frac{(x \mapsto \delta) \in \Gamma}{\Gamma \vdash x \rhd \delta, \emptyset}$$

$$\text{COp}\ \frac{\begin{array}{c}\Gamma \vdash e \rhd \delta, C \qquad \Gamma \vdash e' \rhd \delta', C' \\ \oplus \in \{+, -, \cup, \backslash\}\end{array}}{\Gamma \vdash e \oplus e' \rhd \delta + \delta', C \wedge C'} \qquad \text{CBOp}\ \frac{\begin{array}{c}\Gamma \vdash e \rhd \delta, C \qquad \Gamma \vdash e' \rhd \delta', C' \\ \oplus \in \{=, <, \&\}\end{array}}{\Gamma \vdash e \oplus e' \rhd 0, C \wedge C' \wedge (\delta = 0 \wedge \delta' = 0)}$$

$$\text{CSel}\ \frac{\begin{array}{c}\Gamma[\overline{x \mapsto 0}] \vdash e \rhd \delta, C \\ \Gamma \vdash e' \rhd \delta', C'\end{array}}{\Gamma \vdash \sigma_{\lambda\overline{x}.e}(e') \rhd \delta', C \wedge C' \wedge (\delta = 0)} \qquad \text{CProj}\ \frac{\Gamma \vdash e \rhd \delta, C}{\Gamma \vdash \Pi_{\overline{x}}(e) \rhd \delta, C} \qquad \text{CProd}\ \frac{\begin{array}{c}\Gamma \vdash e \rhd \delta, C \\ \Gamma \vdash e' \rhd \delta', C'\end{array}}{\Gamma \vdash e \times e' \rhd \delta \times \delta', C \wedge C'}$$

Fig. 6: Bound Constraint Derivation

rule CVal assigns the delta value zero to values with no constraints. (Values are exact.) The rule CVar retrieves the bindings for delta variables from the environment. The rule COp states that the delta for the result of the operators $\{+, -, \cup, \backslash\}$ is the sum of the delta of its operands. On the other hand, the rule CBOp requires the operands of the boolean operators $\{=, <, \&\}$ to be exact and states that the result is exact as well. We elide the similar rule for the unary negation operator !. The rule CSel requires the selection condition to be exact and states that the delta of the resulting relation is the same as the input relation. In other words, the resulting relation is stale by the same number of elements as the input relation. Similarly, the rule CProj states that the delta of the resulting relation is the same as the input relation. On the other hand, the rule CProd states that the delta for the resulting relation is the multiplication of the deltas for the input relations. In our running example, let us associate the bound variables drs and dms to rs and ms respectively. The constraint inferred for querySpace is $dms \leq \epsilon_2$, for queryReservations is $drs \leq \epsilon_1$, and for querySpace that involves the join operator (product and selection) is $drs \times dms \leq \epsilon_3$. More detailed explanation for these derivation is available in the appendix [5].

We now define the notion of sufficiently-recent states. Intuitively, a state is sufficiently-recent with respect to the target state if the difference of the return value of every method call on that state versus the target state is within the declared bound of the method.

Definition 12 (Sufficiently-recent State). *A state $\langle v_1, .., v_n\rangle$ is a sufficiently-recent state with respect to the target state $\langle v_1^*, .., v_n^*\rangle$ for an object o iff for every method* def $\epsilon\ m(x)(\langle \sigma_1, .., \sigma_n\rangle)\ \langle e_g, e_u, e_r\rangle$ *of o, and every argument v, let v_r be $\left[\!\left[e_r[x \mapsto v]\overline{[\sigma_i \mapsto v_i]} \right]\!\right]$ and v_r^* be $\left[\!\left[e_r[x \mapsto v]\overline{[\sigma_i \mapsto v_i^*]} \right]\!\right]$, we have $\Delta(v_r, v_r^*) \leq \epsilon$.*

The following lemma states that the bound inference presented in Fig. 6 is sound. In other words, if the inference derives the constraints C for an object, for any solution S of C, if the staleness of each state element σ_i of the object remains within the bound $S(d\sigma_i)$, then the state remains sufficiently-recent.

Lemma 4 (Soundness of Bound Inference). *Given an object o with the state variables $\langle \sigma_1, .., \sigma_n \rangle$, if $o \rhd C$ that is the constraints C (over the bound variables $\overline{d\sigma_i}$) are derived for o, and S is a solution for C, then for every pair of states $\sigma = \langle v_1, .., v_n \rangle$ and $\sigma^* = \langle v_1^*, .., v_n^* \rangle$, if $\overline{\Delta(v_i, v_i^*) < S(d\sigma_i)}$ then σ is sufficiently-recent for σ^*.*

There may be many solutions for the derived constraints, and hence, many sound state bounds that preserve the user-specified bounds for the object. However, solutions that allow more staleness (albeit appropriately bounded) are more favorable since they allow more buffered calls and require less communication. Thus, a candidate objective function to maximize is $d\sigma_1 + .. + d\sigma_n$. In other words, what are the largest delta bounds for the state elements that still preserve the recency specifications of the methods? This function gives the same weight to all the state elements; however, some may be updated more frequently. Let f_i be the relative update frequency of the state element σ_i. Frequencies can be obtained from historical logs or profiling. The objective function is defined as the following weighted sum $d\sigma_1/f_1 + .. + d\sigma_n/f_n$. More frequently updated state elements are given proportionally larger bounds. In our running example, let $\epsilon_1 = 3$, $\epsilon_2 = 4$, and $\epsilon_3 = 6$. If the update frequency of rs is twice as ms, the optimum solution is $drs = 3$ and $dms = 2$.

Definition 13 (Recency Bound Optimization). *Give an object o and the relative update frequency f_i of the state elements σ_i of o, if $o \rhd C$ then the optimum staleness bounds for o are the solution S of C that maximizes $d\sigma_1/f_1 + .. + d\sigma_n/f_n$.*

It is obvious that the objective function can be easily translated to a linear function by multiplying the least common denominator of the frequencies.

6 The Power and The Protocol of Recency-aware Objects

Now, we show that recency-aware objects are stronger than the perfect failure detector abstraction [18] and present a protocol that implements recency-aware objects using perfect failure detectors. These two results show that that recency-aware objects have the same computational power as the perfect failure detector.

The perfect failure detector abstraction $\mathcal{P}$ notifies processes about the crash of the other processes in a synchronous network. It has the following properties: Liveness: Every crashed process is eventually detected by all correct processes. Safety: No correct process is ever suspected by other processes. The recency-aware object $\mathcal{R}$ has the following liveness and safety properties. Liveness: If the user makes a request to a correct replica, it eventually responds. Safety: Executed calls that are yet pending for each correct replica is bounded. The following lemma states that $\mathcal{P}$ is reducible to $\mathcal{R}$ and also its opposite, $\mathcal{R}$ is reducible to $\mathcal{P}$.

```
RecencyAwareObject
  request: call(C)
  indication: ret(C, V) | aborted(C)
  Params:
    $\epsilon$: Int
    SConf: Set[$\mathcal{M}$]
  Using:
    $rb$: ReliableBroadcast
    $pl$: PerfectPointToPointLink
    $pfd$: Perfect Failure Detector
    $bro$: BasicRepObject
  State:
    $\sigma: \Sigma = \sigma_0$; $buff = \emptyset$; $wq = \emptyset$;
    $up = \mathcal{N}$;  $p : \mathcal{N} \mapsto$ Set[C] $= \mathcal{N} \rightarrow \emptyset$

request (call($c$))    if (method($c$) $\notin$ $blocked(bro)$)
  if ($\neg\mathcal{P}(\sigma, c)$)
    issue indication aborted($c$)
  else
    if (method($c$) $\notin$ SConf $\wedge$
          InvSuff($buff$) $\wedge$ LetPRComm($buff$))
      foreach ($r \in up \setminus \{$self$\}$)
        $p'(r) \leftarrow ((p(r) \setminus \{buff\}) \cup \{c \cdot buff\})$
      if (InBound($p'$))
        $p \leftarrow p'$
        exec($c$);  $buff \leftarrow c \cdot buff$
      else
        issue request ($rb$, broadcast(buff($buff$)))
        insert($wq$, $c$)
    else
      foreach ($r \in up \setminus \{$self$\}$)
        $p'(r) \leftarrow (p(r) \cup \{c\})$
      if (InBound($p'$))
        $p \leftarrow p'$
        issue request($bro$, call($c$))
      else
        issue request ($rb$, broadcast(buff($buff$)))
        insert($wq$, $c$)
indication crash($pfd$, $p$)
  $up \leftarrow up \setminus \{p\}$
fun InBound($p$)
  foreach($n \in up$)
    if ($\sum_{c' \in p(n)}$ weight($c'$) $> \epsilon/(\mathcal{N} - 1)$)
      return False
  return True
indication ($rb$, deliver($n$, buff($buff$)))
  if (self $\neq n$)
    exec($buff$)
    issue request ($pl$, send($n$, ack($buff$)))
indication ($pl$, deliver($n$, ack($c$)))
  $p \leftarrow p[n \mapsto (p(n) \setminus \{c\})]$
  foreach ($c \in wq$) issue request(call($c$))
  $wq \leftarrow \emptyset$
fun exec($c$)
  $\sigma \leftarrow$ update($c$)($\sigma$);  $v \leftarrow$ retv($c$)($\sigma$)
  issue indication ret($c$, $v$)
indication ($bro$, ret($c$, $v$))
  issue request ($pl$, send(orig($c$), ack($c$)))
  issue indication ret($c$, $v$)
```

Fig. 7: Recency-Aware Protocol

Lemma 5. $\mathcal{P} \preceq \mathcal{R} \ \wedge \ \mathcal{R} \preceq \mathcal{P}$.

For the proof of the first conjunct, consider two replicas rep_1 and rep_2. We show by contradiction that rep_1 will eventually know whether rep_2 has crashed. We assume the opposite. Consider an execution where rep_1 has already executed a set of requests R and receives another request r from the user, such that the pending set $R \cup \{r\}$ makes a difference in the state of rep_2 that pushes it out-of-bound. By the contradiction assumption, rep_1 is never informed when rep_2 crashes. Therefore, if rep_1 does not hear from rep_2, the following two scenarios are indistinguishable to rep_1. (S_1) The replica rep_2 has crashed. (S_2) The replica rep_2 is too slow. The replica rep_1 has the following two choices: (C_1) The replica rep_1 waits to hear from rep_2 about receiving a request in R before processing and responding to r. (C_2) The replica rep_1 processes and responds to r. If the protocol makes the choice C_1, it might be the scenario S_1 and then the liveness property is violated. If the protocol makes the choice C_2, it might be the scenario S_2 and then the recency bound for rep_2 is violated.

The second conjunct, directly follows from the protocol. We briefly describe the protocol in Fig. 7 that implements a recency-aware replicated object. The full description of the protocol is available in the appendix [5]. Given an object definition, the protocol benefits from both static and dynamic coordination analysis to guarantee convergence, integrity and recency. To reduce communication, replicas try to execute the calls locally while maintaining the staleness bound

ϵ. Each replica keeps its locally executed calls in a buffer *buff* before they are broadcast. Replicas send an acknowledgement ack to the originating replica once they receive and execute a call or a buffer of calls. Each replica *rep* keeps a map called pending p from each replica rep' to the set of pending calls sent from *rep* to rep'. When a replica originates a call c, it adds c to its local pending set for each of the other replicas; once it receives an acknowledgement for c from a replica rep', it removes c from the set of pending calls for rep'. Each replica keeps the set of correct replicas *up*, and removes a replica from the set if the prefect failure detector *pfd* issues a crash event for that replica. A requested call can be executed only if it does not push the pending set for any correct replica out of the bound. Otherwise, it cannot be immediately executed and is kept in a waiting queue *wq* to be retried later, and further, the buffer is sent to the other replicas and is reset to accelerate the shrinking of the pending set. To decide whether a call can be executed locally, the conditions of the rule CallLocal of the operational semantics (§ 4) are checked. The set of state-conflicting methods SConf that is statically calculated is consulted to check if the call is all-state-commutative. The validity of the two conditions invariant-sufficiency and let-$\mathcal{P}$-R-commutativity of the buffer (after the new call is added) are dynamically decided by a solver at run-time. If the conditions do not hold, the call is coordinated with other replicas using the basic blocking coordination protocol *bro* [30] that guarantees integrity and convergence but not recency.

7 Experimental Results

We have implemented the analysis and protocol as a synthesis tool called Hampa. We applied it to two use-cases: the bank account use-case (with the withdraw, deposit and balance methods and the integrity property of non-negative balance) and the movie booking use-case (Fig. 2). The experiments show that as the staleness bound increases, the coordination overhead and response time of recency-aware objects is decreased. Further, recency-aware objects are twice as responsive as sequentially consistent counterparts.

Platform and setup. The experiments are conducted on a cluster of 4 computing nodes. Each node has 2 AMD Opteron 6272 CPUs with a total 8 cores, 64GB ECC memory and 40Gbps InfiniBand network. JDK is openjdk version 1.8.0_222. We used the CVC4 [11] SMT solver v.1.7. Reported numbers are the arithmetic means of results from three repetitions on 4 replicas. In the experiments for the bank account use-case, all the calls are applied to the same account object and the amount is selected randomly in the range [10, 20]. For the movie use-case, we send requests for each movie identifier to the same replica. Further, we do not issue offScreen calls because taking a movie off-screen causes later method calls on the same movie to be aborted and thus, these methods are not fully exercised. This would significantly improve the response time. However, in practice, offScreen calls are rarely used. The movie and user IDs are chosen at random from six and a hundred unique IDs. In all the experiments, we execute 500 calls in millisecond intervals evenly distributed between 4 replicas.

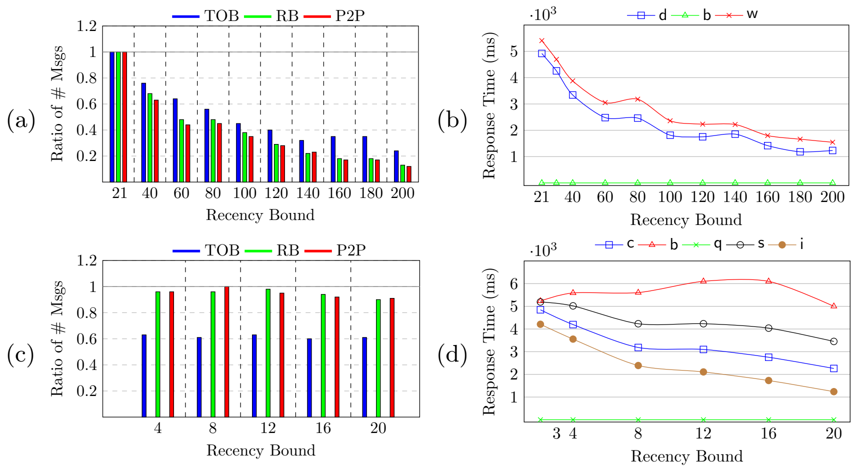

Fig. 8: Effect of recency on coordination load and response time. (a) and (b) show the bank account use-case. d, w, and b stand for deposit, withdraw and balance (with the frequencies of 75%, 25%, 5% in the workload respectively). (c) and (d) show the movie use-case. c, b, q, s, and i stand for cancelBook, book, querySpace, specialReserve, and increaseSpace (with frequencies 4%, 6%, 5%, 40% and 45%).

Measurements. We measure two comparison criteria: coordination load and response time. At the lower layers, the protocol reduces to three communication primitives: total-order-broadcast (TOB), reliable-broadcast (RB) and point-to-point links (P2P). To measure the coordination overhead, we separately count the number of different types of messages that replicas send during the the execution of their requests. The response time for a call is the duration between the time that the client requests the call and the time that the user receives the return value.

We performed three experiments. In the first experiment, we study the effect of increasing the staleness bound on the coordination load. We report the ratio of the number of messages that the protocol sends for the bound under test over the number of messages that it sends for the base-line bound. (The base-line recency bound is the maximum weight of the calls. The baseline allows every single call to be buffered.) In the second experiment, we study the effect of increasing the staleness bound on the response time of each method. Finally, in the last experiment, we compare the response time of our protocol with the base-line recency, with the sequential consistency (SC). SC uses total-order broadcast for all the methods.

Assessment. Fig. 8.(a) and (c) show the effect of increasing the staleness bound on the coordination load for the two use-cases. As the staleness bound is increased, the ratio of the messages sent by RB, TOB and P2P decreases. Fig. 8.(a) (bank account), shows 88% decrease in the number of messages sent to RB when the bound is increased from 20 to 200. Likewise, the TOB and P2P ratios decrease by 78% and 90%, respectively. In Fig. 8.(c) (movie booking), buffering helps to reduce TOB calls by 40% across the experiments. This decrease, however, unlike the bank account use-case, is steady over different bounds. This is because it is more difficult to "buffer" in the movie booking use-case. There are no $\mathcal{S}$-conflicts in the bank account use-case and hence two out of two update methods can be buffered. However, $\mathcal{S}$-conflicts in the movie use-case allow only 2 out of 4 update methods to be buffered: increaseSpace and specialReserve. Also, we observe that the number of RB and P2P messages decrease by at most 10%.

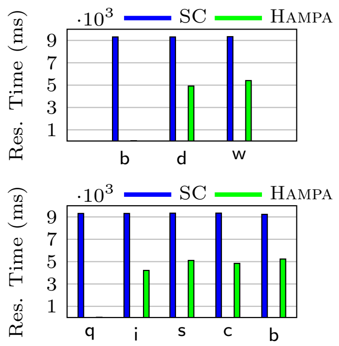

Fig. 9: Response time comparison between Hampa and sequential consistency for each method type. Top: bank account, Bottom: movie booking use-case.

Fig. 8.(b) and (d) shows the effect of increasing the staleness bound on the response time for the two use-cases. In Fig. 8.(b) (bank account), the response time of withdraw and deposit methods decrease by 71% and 75%, respectively when the staleness bound is increased from 20 to 200. The withdraw method is the least responsive method. The reason is that it has a self-conflict and requires synchronization if it cannot be buffered. In Fig. 8.(d) (movie booking), we observe slight increase in response time for the book method while increasing the bound from 2 to 20. This is because the book operation cannot be buffered due to the $\mathcal{S}$-conflict with other methods and has to be synchronized. On the other hand, the response time of the specialReserve method decreases by 33% when the bound is increased from 2 to 20. The reason is that it has a self-conflict and if it cannot be buffered, it should be synchronized by the TOB and TOB incurs a high coordination overhead. Therefore, as buffered calls increase and the use of TOB decreases, the response time is significantly improved. The response time of the increaseSpace method also benefits from recency awareness; it decreases by 72%. The methods book and cancelBook have conflicts. In the blocking protocol that Hampa uses, the method book handles synchronization; therefore, the method cancelBook just broadcasts the request. As the recency bound is increased, the network is less crowded and therefore, the response time of cancelBook is decreased.

Fig. 9 compares the response time of recency-aware objects with the baseline bound with the sequentially consistent objects. The SC protocol synchronizes all the calls and orders them with respect to each other. However, Hampa minimizes coordination while preserving convergence, integrity and recency. We observe that the response time speedup is in average as high as $2\times$ and $1.8\times$ for the bank account and movie use-cases respectively. More experiments are available in the appendix [5]. In particular, they show that the runtime cost of SMT solving is only 0.2% to 1% of the average response time.

8 Related Work

Epsilon serializability [46] allows concurrent execution of updates with queries and bounds the difference of the inconsistent values that are observed in these executions and the consistent values that would be observed in a serializable execution. In contrast, HAMPA preserves the integrity of the state, bounds staleness, allows different orders in different replicas, and formally defines the difference for relational operators.

In TACT [56,55,54,57,58], operations return tentative values; they might be eventually reordered to preserve strong consistency. TACT bounds the numeric error between the tentative and final return values. The user specifies the granularity of the bounded object "conit" and the strength of the protocol. On the other hand in HAMPA, the states are final and enjoy integrity provided on top of weak consistency. Further, the staleness bound with respect to the pending future state is automatically optimized with static and dynamic analyses.

In AQuA [31], given a query and a staleness bound, the master server dynamically selects a recent enough server to service the query. Similarly, TRAPP [43] finds recent enough servers for different parts of data that are needed for the query. FRACS [59] allows operations to be buffered at replicas up to a given threshold. In contrast to HAMPA, these projects do not guarantee integrity and convergence, and do not automatically infer the staleness bounds. PIQL [6] bounds the number of key-value store operations for each query trading the precision of the result for performance. However, it does not consider the staleness of replicas.

To reduce synchronization, PBS [9] communicates with only a partial quorum of replicas to bring a total order to operations, and probabilistically bounds the staleness of the observed states. In contrast, HAMPA performs synchronization with full quorums but only for conflicting calls, and allows different orders for replicas. Further, it analyzes and synthesizes replicated objects and supports relational in addition to single-key operations.

The trade-off between consistency and latency presented as PACELC [1] aligns with our experiments. As the consistency decreases (staleness bound increases), the latency decreases (responsiveness increases). Warranties [38] and Homeostasis [47] allow local updates if they keep the validity of certain assertions. Although other replicas can rely on the validity of the assertions, the staleness of their state is not bounded. In contrast, HAMPA maintains a staleness bound. Further, it exploits weak consistency and guarantees convergence.

9 Conclusion

This paper presented a relational object specification language that captures the integrity and recency requirements of the object. It presented a syntax-directed analysis that given a specification, infers optimum staleness bounds. In addition, it presented the coordination avoidance conditions, operational semantics, a protocol and a synthesis tool for replicated systems that guarantee convergence, integrity and recency. The recency-aware protocol embeds a solver to decide whether coordination avoidance is safe and increases the responsiveness.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design. Computer **45**(2) (2012)
2. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Computing **9**(1) (1995)
3. Alagappan, R., Ganesan, A., Lee, E., Albarghouthi, A., Chidambaram, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Protocol-aware recovery for consensus-based distributed storage. ACM Transactions on Storage (TOS) **14**(3), 1–30 (2018)
4. Alglave, J., Cousot, P., Maranget, L.: Syntax and semantics of the weak consistency model specification language cat. arXiv preprint arXiv:1608.07531 (2016)
5. Appendix: Appendix. Attached at the paper. (2020)
6. Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M.J., Patterson, D.A.: Piql: Success-tolerant query processing in the cloud. Proceedings of the VLDB Endowment **5**(3), 181–192 (2011)
7. Bailis, P., Fekete, A., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Coordination avoidance in database systems. Proc. VLDB Endow. **8**(3), 185–196 (Nov 2014). https://doi.org/10.14778/2735508.2735509, http://dx.doi.org/10.14778/2735508.2735509
8. Bailis, P., Fekete, A., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Feral concurrency control: An empirical investigation of modern application integrity. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1327–1342. ACM (2015)
9. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. Proceedings of the VLDB Endowment **5**(8), 776–787 (2012)
10. Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguica, N., Najafzadeh, M., Shapiro, M.: Putting consistency back into eventual consistency. In: Proceedings of the Tenth European Conference on Computer Systems. pp. 6:1–6:16. EuroSys '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2741948.2741972, http://doi.acm.org/10.1145/2741948.2741972
11. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11). Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (Jul 2011), http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf, snowbird, Utah
12. Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., Zheng, J.: PRACTI replication. In: Proc. NSDI (2006)
13. Birman, K.P.: Replication and fault-tolerance in the ISIS system. In: Proc. SOSP (1985)
14. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: Automated conflict-free distributed implementation of component-based models. In: International Symposium on Industrial Embedded System (SIES). pp. 108–117. IEEE (2010)
15. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Proc. POPL (2014)
16. Bouajjani, A., Enea, C., Mukund, M., Shenoy, G., Suresh, S.: Formalizing and checking multilevel consistency. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 379–400. Springer (2020)

17. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: Specification, verification, optimality. In: Proc. POPL (2014)
18. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming. Springer Publishing Company, Incorporated, 2nd edn. (2011)
19. Clancy, K., Miller, H.: Monotonicity types for distributed dataflow. In: Proceedings of the Programming Models and Languages for Distributed Computing. p. 2. ACM (2017)
20. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. **1**(2) (2008)
21. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. **31**(3), 8:1–8:22 (Aug 2013). https://doi.org/10.1145/2491245, http://doi.acm.org/10.1145/2491245
22. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proc. SOSP (2007)
23. Drăgoi, C., Henzinger, T.A., Zufferey, D.: Psync: a partially synchronous language for fault-tolerant distributed algorithms. ACM SIGPLAN Notices **51**(1), 400–415 (2016)
24. Emmi, M., Enea, C.: Monitoring weak consistency. In: Proc. CAV (2018)
25. Emmi, M., Enea, C.: Weak-consistency specification via visibility relaxation. Proceedings of the ACM on Programming Languages **3**(POPL), 1–28 (2019)
26. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (Apr 1985). https://doi.org/10.1145/3149.214121, http://doi.acm.org/10.1145/3149.214121
27. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2) (June 2002)
28. Gilbert, S., Lynch, N.A.: Perspectives on the CAP theorem. IEEE Computer **45**(2), 30–36 (2012)
29. Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: '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. pp. 371–384. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837625, http://doi.acm.org/10.1145/2837614.2837625
30. Houshmand, F., Lesani, M.: Hamsaz: Replication coordination analysis and synthesis. In: Proceedings of Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '19, ACM, New York, NY, USA (2019)
31. Krishnamurthy, S., Sanders, W.H., Cukier, M.: An adaptive quality of service aware middleware for replicated services. IEEE Transactions on Parallel and Distributed Systems **14**(11), 1112–1125 (2003)
32. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. Comput. Syst. **10**(4) (1992)
33. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (1978)
34. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2) (1998)

35. Lewchenko, N.V., Radhakrishna, A., Gaonkar, A., Černỳ, P.: Conflict-aware replicated data types. arXiv preprint arXiv:1802.08733 (2018)
36. Li, C., Leitão, J.a., Clement, A., Preguica, N., Rodrigues, R., Vafeiadis, V.: Automating the choice of consistency levels in replicated systems. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. pp. 281–292. USENIX ATC'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2643634.2643664
37. Li, C., Porto, D., Clement, A., Gehrke, J., Preguica, N., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. pp. 265–278. OSDI'12, USENIX Association, Berkeley, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2387880.2387906
38. Liu, J., Magrino, T., Arden, O., George, M.D., Myers, A.C.: Warranties for faster strong consistency. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. pp. 503–517. NSDI'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2616448.2616495
39. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In: Proc. SOSP (2011)
40. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: Proc. NSDI (2013)
41. Nair, S., Petri, G., Shapiro, M.: Proving the safety of highly-available distributed objects. In: Proc. ESOP (2020)
42. Oki, B.M., Liskov, B.H.: 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. pp. 8–17. PODC '88, ACM, New York, NY, USA (1988). https://doi.org/10.1145/62546.62549, http://doi.acm.org/10.1145/62546.62549
43. Olston, C., Widom, J.: Offering a precision-performance tradeoff for aggregation queries over replicated data. Tech. rep., Stanford (2000)
44. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference. pp. 305–320. USENIX ATC'14, USENIX Association, Berkeley, CA, USA (2014), http://dl.acm.org/citation.cfm?id=2643634.2643666
45. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible update propagation for weakly consistent replication. In: Proc. SOSP (1997)
46. Ramamritham, K., Pu, C.: A formal characterization of epsilon serializability. IEEE Transactions on Knowledge and Data Engineering **7**(6), 997–1007 (1995)
47. Roy, S., Kot, L., Bender, G., Ding, B., Hojjat, H., Koch, C., Foster, N., Gehrke, J.: The homeostasis protocol: Avoiding transaction coordination through program analysis. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1311–1326. SIGMOD '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2723372.2723720, http://doi.acm.org/10.1145/2723372.2723720
48. Shapiro, M., Preguica, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Tech. Rep. RR-7506, INRIA (2011)
49. Sivaramakrishnan, K., Kaki, G., Jagannathan, S.: Declarative programming over eventually consistent data stores. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 413–424. PLDI '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737981, http://doi.acm.org/10.1145/2737924.2737981

50. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 385–400. SOSP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2043556.2043592, http://doi.acm.org/10.1145/2043556.2043592
51. Terry, D.B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M.K., Abu-Libdeh, H.: Consistency-based service level agreements for cloud storage. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 309–324. SOSP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2517349.2522731, http://doi.acm.org/10.1145/2517349.2522731
52. Vogels, W.: Eventually consistent. ACM Queue **6**(6) (2008)
53. Wang, C., Enea, C., Mutluergil, S.O., Petri, G.: Replication-aware linearizability. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 980–993 (2019)
54. Yu, H., Vahdat, A.: 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. p. 21. USENIX Association (2000)
55. Yu, H., Vahdat, A.: Efficient numerical error bounding for replicated network services. In: VLDB. pp. 123–133. Citeseer (2000)
56. Yu, H., Vahdat, A.: Combining generality and practicality in a conit-based continuous consistency model for wide-area replication. In: Proceedings 21st International Conference on Distributed Computing Systems. pp. 429–438. IEEE (2001)
57. Yu, H., Vahdat, A.: The costs and limits of availability for replicated services. In: ACM SIGOPS Operating Systems Review. vol. 35, pp. 29–42. ACM (2001)
58. Yu, H., Vahdat, A.: Minimal replication cost for availability. In: Proceedings of the twenty-first annual symposium on Principles of distributed computing. pp. 98–107. ACM (2002)
59. Zhang, C., Zhang, Z.: Trading replication consistency for performance and availability: an adaptive approach. In: Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on. pp. 687–695. IEEE (2003)