Hambazi: Spatial Coordination Synthesis for Augmented Reality

YI-ZHEN TSAI, University of California, Riverside, USA
JIASI CHEN, University of Michigan, USA
MOHSEN LESANI, University of California, Santa Cruz, USA

Augmented reality (AR) seamlessly overlays virtual objects onto the real world, enabling an exciting new range of applications. Multiple users view and interact with virtual objects, which are replicated and shown on each user's display. A key requirement of AR is that the replicas should be quickly updated and converge to the same state; otherwise, users may have laggy or inconsistent views of the virtual object, which negatively affects their experience. A second key requirement is that the movements of virtual objects in space should preserve certain integrity properties either due to physical boundaries in the real world, or privacy and safety preferences of the user. For example, a virtual cup should not sink into a table, or a private virtual whiteboard should stay within an office. The challenge tackled in this paper is the coordination of virtual objects with low latency, spatial integrity properties and convergence. We introduce "well-organized" replicated data types that guarantee these two properties. Importantly, they capture a local notion of conflict that supports more concurrency and lower latency. To implement well-organized virtual objects, we introduce a credit scheme and replication protocol that further facilitate local execution, and prove the protocol's correctness. Given an AR environment, we automatically derive conflicting actions through constraint solving, and statically instantiate the protocol to synthesize custom coordination. We evaluate our implementation, Hambazi, on off-the-shelf Android AR devices and show a latency reduction of 30.5-88.4% and a location staleness reduction of 35.6-75.6%, compared to three baselines, for varying numbers of devices, AR environments, request loads, and network conditions.

CCS Concepts: • **Theory of computation → Distributed computing models**; • **Software and its engineering → Distributed systems organizing principles**; • **Human-centered computing → Mixed / augmented reality**.

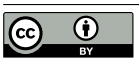
Additional Key Words and Phrases: Multi-user Augmented Reality, Distributed Coordination, Coordination Avoidance, Fault Tolerance, Well-Organization

ACM Reference Format:
Yi-Zhen Tsai, Jiasi Chen, and Mohsen Lesani. 2025. Hambazi: Spatial Coordination Synthesis for Augmented Reality. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 91 (April 2025), 30 pages. https://doi.org/10.1145/3720425

1 Introduction

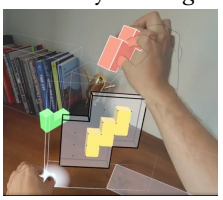
Augmented reality (AR) is one of the key technologies driving the next generation of mobile applications. AR allows users to view virtual objects overlaid on top of the real world, with applications in entertainment (*e.g.,* Pokemon Go), workspaces (*e.g.,* Apple Vision Pro), education (*e.g.,* spatial understanding [1]), and public safety (*e.g.,* firefighting [19]), as illustrated in Fig. 1. Industry has made huge investments in the space, with Apple announcing its own AR headset in June 2023. In multi-user AR applications, *multiple users view and interact with the same set of virtual*

Authors' Contact Information: Yi-Zhen Tsai, University of California, Riverside, Riverside, USA, ytsai036@ucr.edu; Jiasi Chen, University of Michigan, Ann Arbor, USA, jiasi@umich.edu; Mohsen Lesani, University of California, Santa Cruz, Santa Cruz, USA, mlesani@ucsc.edu.

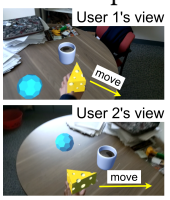


ACM 2475-1421/2025/4-ART91
https://doi.org/10.1145/3720425

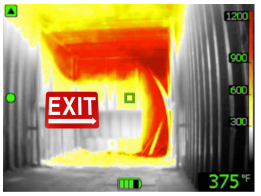
objects simultaneously. For example, two users could attempt to interact with a virtual cheese at the same time, and move the cheese in opposite directions, as shown in Fig. 1b. User devices need to *coordinate* on the position of the virtual objects, and then quickly render them for all users. The replicas of a virtual object should eventually *converge* to the same position.



(a) Education [1]: Users collaboratively complete 3D puzzles.



(b) Fun: Users simultaneously move holograms.



(c) Public safety: Firefighters move the virtual "exit" sign for safe navigation.

Fig. 1. Examples of multiple AR users interacting with virtual objects.

Unfortunately, coordination and communication delays arise in practice, preventing multiple users from quickly observing updates to the virtual objects. Glitches, jumps and rollbacks can affect AR user experience. In order to provide the AR users with the desirable quality-of-experience [4, 21], AR demands *low latency* between the time a user interacts with a virtual object to the time it is updated on the display. Further, a key twist arises from the AR context: virtual objects are overlaid onto the real world, and should interact with the real world in physically meaningful ways. In particular, virtual objects are expected to maintain an *integrity* property: they should never enter *restricted zones*. Examples include physical boundaries (*e.g.,* a virtual billiards ball should not enter a wall). Restriction zones can be further imposed by the user or by the system, such as by user's privacy policies (*e.g.,* a private virtual whiteboard should not be moved outside an office [82]), by safety and security policies (*e.g.,* a virtual object should not leave a user-drawn safe playing area [62]), or by reliability policies (*e.g.,* preventing occlusion of critical information in a dangerous area during human-robot interactions [89]). These restriction zones could also be moving (*e.g.,* a virtual object should not occlude a pedestrian walking in the user's field of view [50]).

Coordination for emerging user-centric applications [20] that satisfies all three requirements – convergence on the position of a virtual object, while respecting the restricted zones, and doing so with low latency – is challenging. ARCore (Google's Android API for AR) records the state of the AR session in a Firestore database [34, 35] and can provide optimistic concurrency control wherein users read the current state, perform a computation and then submit a write. In the interim, if another user made an update that invalidated the read values, the write is re-tried a fixed number of times before failing. However, this approach incurs significant latency from two sources: (a) communication latency to an edge/cloud server can be hundreds of milliseconds; (b) when multiple users contend to update the same state, it leads to high failure rates and multiple re-tries, increasing latency. Another approach is "netcode", deployed by major game engines to manage player positions, hit points, etc. [31, 58, 90, 91]. Essentially, users optimistically execute their actions locally and then reconcile their actions with the server, and roll back if consistency issues arise. The major issue with this approach is that rollback leads to poor user experience (*e.g.,* the virtual billiards ball will retrace its path).

In this paper, we take a principled look at multi-user interactions with virtual objects in AR. We study the following question: ***can we automatically construct protocols that coordinate the interactions of multiple users on virtual objects with low-latency and convergence, and without entering restricted zones and rollbacks?*** Our main design philosophy is *hybrid consistency* [7–9, 26, 36, 40, 41, 51–55, 87], allowing local actions without coordination in order to meet low latency goals, and coordinating when necessary in order to provide convergence and integrity (*i.e.,* avoiding restricted zones). We then introduce the notion of *well-organized replicated data types* that guarantee convergence and integrity. Well-organization is inspired by

well-coordination [40] and allows processes to make local updates without coordination wherever possible. Although some users' views may be temporarily stale, they eventually converge. Compared to well-coordination, well-organization introduces two new key ideas that reduce latency: the notion of conflict is defined locally considering the current state rather than globally for every state, and it does not track and propagate dependencies.

To preserve integrity and convergence, well-organization requires certain conditions for the execution and propagation of actions; in particular, conflicting actions should have the same order across processes. We introduce *a credit scheme and replication protocol* that implements well-organized replicated data types for AR applications, and prove that the implementation is correct. Each action needs to acquire enough credit to prevent its conflicting actions. Importantly, if the current user's process already has enough credit, it does not need to communicate with any other users' processes to avoid conflicts, thus preserving integrity without global synchronization. One challenge is that users can launch AR applications in arbitrary environments with unique restricted zones. The protocol is parametric in terms of these conflicts. We formally define conflicts, automatically calculate conflicting actions using off-the-shelf constraint solvers, and then statically instantiate the protocol with the pre-computed conflicting actions. In other words, we *synthesize a custom protocol for a given AR environment.*

To evaluate our protocol, we implement it in a system called Hambazi on Google ARCore Android devices. We compare Hambazi with three baseline methods (well-coordination, netcode principles from game design, and Google Firestore) and demonstrate up to 88.4% reduction in average latency, and 75.6% reduction in location staleness. In summary, the main contributions of this paper are:

- Problem formulation of spatial coordination of multi-user AR applications.
- Well-organized replicated data types that guarantee convergence and integrity (§ 3).
- Spatial coordination protocol that implements well-organization without synchronization, with a fault tolerance mechanism, and proof of correctness (§ 4).
- Implementation of Hambazi that synthesizes correct-by-construction coordination for given AR environments, and its empirical evaluation on Android devices (§ 5).

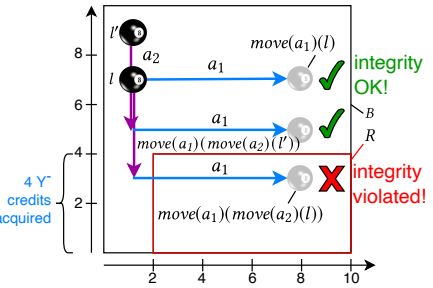
Next, we start with an overview with an example and high-level intuitions.

2 Overview

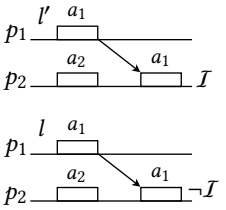
Multi-user AR preliminaries. Consider a multi-user AR application that keeps the current location l of a virtual object. For example, Fig. 2a shows a virtual ball on a real world billiards table. The spatial integrity property $\mathcal{I}$ for the location of the virtual object is to stay within an area B, and not enter a restricted zone R. In our example in Fig. 2a, B is the whole rectangular area, and R is the red restricted zone in the right-bottom corner. A user can call the method $move(a)$ at a process to move the object, which is communicated to the other processes. The action a has a direction d and magnitude m where the directions are $\mathcal{D} = \{X^+, X^-, Y^+, Y^-\}$, corresponding to right, left, up, and down respectively. (Our experiments in § 5 consider more general 3D use-cases.) (When clear from the context, we use method "call" and "action" interchangeably.) The replicated object should preserve the above integrity property, and the states of all processes should eventually converge.

Conflicting actions and well-organization. Consider the example execution in Fig. 2a. Process p_1 (blue color) executes $move(a_1)$ which pushes the AR object to the right. The call $move(a_1)$ is permissible; the resulting location $move(a_1)(l)$ satisfies the integrity property as it is within B and outside of R. Simultaneously, process p_2 (purple color) executes the move $move(a_2)$ without p_1's knowledge, pushing the AR object downwards. From p_2's point of view, a_2 is also permissible. Now, if $move(a_1)$ propagates to process p_2 and is executed there, the resulting final location

$move(a_1)(move(a_2)(l))$ is in the restricted zone R, and thus, violates the integrity property. This is an example of what we call a *permissible-right conflict* of the *move* method with itself: as Fig. 2b (bottom diagram) shows, at a location l, the method call $move(a_1)$ is permissible; however, when it moves to the right of $move(a_2)$, it is not permissible anymore, *i.e.,* violates integrity ($\neg\mathcal{I}$). As Fig. 2b shows (top diagram), starting from the pre-state location l', when $move(a_1)$ moves to the right of $move(a_2)$, it stays permissible, *i.e.,* preserves integrity ($\mathcal{I}$).



(a) Conflicting actions.



(b) At l', a_1 permissible-right commutes with a_2. At l, a_1 permissible-right conflicts with a_2.

Fig. 2. Conflicting actions in an AR application. The integrity property requires the ball to not enter the red area. At l, action a_1 is not right-commutative with a_2: if a_1 occurs after a_2, the ball enters the red area.

To coordinate replicated virtual objects, we propose a new hybrid consistency model called *well-organization* that builds on prior work on well-coordination [40], with key differences. Well-coordination requires conflicting calls to be synchronized with each other (using a total order broadcast) while other calls can proceed without synchronization. However, well-coordination is overly conservative in its notion of conflict. In brief, well-coordination defines conflict globally no matter the state: if there exists *any* pre-state σ where a call c_1 permissible-right conflicts with another call c_2, then c_1 conflicts with c_2. In the example above, at the initial pre-state l, the two calls $move(a_1)$ and $move(a_2)$ conflict. Thus, according to well-coordination, they are considered conflicting globally in the application. This leaves no room for concurrency between them (even at pre-state l'); they should have the same order across all processes. In contrast to well-coordination, in this work, we observe that from a pre-state to another, the set of conflicting calls can be different, and introduce well-organization that defines the notion of conflict locally for each pre-state. In the example in Fig. 2a, starting from the higher pre-state l', if $move(a_1)$ propagates to process p_2 and moves right to after $move(a_2)$, it still satisfies integrity ($\mathcal{I}$). Thus, at the pre-state l', $move(a_1)$ does not conflict with $move(a_2)$, and processes p_1 and p_2 can execute $move(a_1)$ and $move(a_2)$ concurrently. Therefore, well-organization supports more concurrency that leads to lower latency. We formally define well-organization with this notion of local conflicts, and further a new treatment of dependencies in § 3, and prove that it guarantees integrity and convergence.

Coordination. Conflicting actions should have the same order across all processes. This is often achieved by broadcasting conflicting actions to a total-order broadcast instance. In contrast, Hambazi avoids coordination and thus reduces latency. It allows processes to execute even conflicting actions locally wherever possible. We propose a credit scheme where each dimension has a total amount of credit corresponding to its length in the boundary B. For example, in Fig. 2a, B has width 10 units, and there are a total of 10 credits across the X^+ (right) and X^- (left) directions. The ownership of credits is equally distributed between processes. A process can transfer credits to another. If a process owns the required credit for an action, it can take the action locally, and simply propagate its update to other processes. For example, in Fig. 2a, a process seeking to move the virtual object right by 7 units must have at least 7 X^+ (right) credits available. Moving in a direction spends credit in that direction, and yields credit in the opposite direction.

Bounded counters [8, 10] similarly escrow values at processes, but their naive application cannot solve the spatial coordination problem. This is because the integrity property of not entering

restriction zones is on the values of all dimensions, and cannot be stated as *independent* invariants on each dimension. For example, in Fig. 2a, the constraint is that $x < 2$ when $y < 4$, and $x < 10$ when $y > 4$. The constraints on x and y are coupled, and a single bound on x, and a single bound on y would fail to capture it. Preserving numeric invariants in a multi-dimensional space is more involved: actions in one dimension can conflict not only with other actions in the same dimension but also actions of other dimensions. In Fig. 2a, we saw that, at l, a_1 in direction X^+ (right) by process p_1 permissible-right conflicts with orthogonal Y^- (down) actions such as a_2 by process p_2. Our key coordination idea is to request and hold sufficient credits to prevent the execution of conflicting actions. Thus, before p_1 takes action a_1 (blue color), it gathers and keeps 4 Y^- credits out of a total of 7 available at that location l. That prevents p_2 (or any other process) from concurrently making the downward action a_2 that needs 4 Y^- credits. Once p_1 is done taking action a_1, it can release those down credits back to other processes. We present the complete protocol and prove that it implements well-organization in § 4.

We formally define conflicts, and given an AR environment, use constraint solvers to statically determine for each location the set of conflicting actions, and the number of credits that should be acquired in each direction to prevent them. The protocol is parametric and is statically instantiated with the solver results to synthesize efficient protocols for the given AR environments.

3 Well-Organized Replicated Data Types

In this section, we introduce well-organized replicated data types. Well-organization is inspired by well-coordination with two novelties. Firstly, in contrast to well-coordination that requires invariant-commutativity globally for all possible pre-states where a call is being executed, well-organization requires it locally only for the current pre-state. This subtle difference allows more concurrency under well-organization. Secondly, we observe that, interestingly, certain checks in well-coordination that track and propagate dependencies are in fact unnecessary. Well-organization provides the same guarantees of convergence and integrity more efficiently.

$$
\begin{array}{rcll}
o & \coloneqq & \langle \Sigma, \mathcal{I}, \overline{u \coloneqq d}, \overline{q \coloneqq d} \rangle & \text{Object} \\
\sigma & : & \Sigma & \text{State} \\
\mathcal{I} & & & \text{Invariant (Integrity)} \\
u & : & U & \text{Update Method} \\
q & : & Q & \text{Query Method} \\
d & : & \lambda x, \sigma.\, e & \text{Definition} \\
e & & & \text{Expression} \\
v & : & V & \text{Value} \\
p & : & P & \text{Process or Replica} \\
r & : & R & \text{Request Identifier} \\
c : C & \coloneqq & u(v)^r_p & \text{Update Method Call} \\
 & & q(v) & \text{Query Method Call} \\
\ell & \coloneqq & \textsc{Call}(p, c) & \text{Label} \\
 & | & \textsc{Prop}(p, c) & \\
 & | & (p, q(v)) & \\
\tau & \coloneqq & \ell^* & \text{Trace}
\end{array}
$$

Fig. 3. Syntax

We first present how an object data type, including its integrity properties, can be simply specified. We then present the core operational semantics for well-organized replicated data types, and prove that it guarantees integrity and convergence. This abstract semantics will serve as the specification for our AR replication protocols in the following section (§ 4).

3.1 Replicated Data Types

In this subsection, we adopt and extend basic definitions from well-coordination [40] that we will use later for the semantics of well-organization.

Data Types. As Fig. 3 shows, a class of objects is a tuple $\langle \Sigma, \mathcal{I}, \overline{u \coloneqq d}, \overline{q \coloneqq d} \rangle$ that defines the state type Σ, the invariant (or integrity property) $\mathcal{I}$ on the state, and the definitions of the update methods u and query methods q. The invariant (or integrity) $\mathcal{I}$ is a predicate on the state (*e.g.*, non-negative balance for a bank account). For example, $\mathcal{I}(\sigma)$ states that the invariant $\mathcal{I}$ holds

for the state σ. The definition of an update method u is a function from the parameter and the pre-state σ to the post-state. An update call c is an update method u applied to an argument value v. Two calls c and c' can be composed $c \circ c'$ with the standard function composition operator $\circ$. Similarly, the definition of a query method is a function from the parameter and the pre-state σ to the return value. The object is replicated on the set of processes P. Clients can request update calls $u(v)$ or query calls $q(v)$ at every process p, and processes coordinate these calls. Calls have unique request identifiers r. An update call is decorated with the issuing process p and the request identifier r. (We omit these decorations when they are not needed or are evident from the context.) A label ℓ for a call request contains the pair of the issuing process, and an update or query call, and a trace τ is a sequence of labels.

Replicated State. The state of the given object is replicated across processes, as shown in Fig. 4. The replicated state ss is a mapping from each process p to its state σ. The execution history x of a process is modeled as a sequence of calls. Since query calls do not mutate the state, an execution history only keeps update calls. We write $c \in x$ to denote that the call c is in the history x. The application $x(\sigma)$ of a history x to a state σ is the application of the composition of the sequence of calls of x to σ. Unique identifiers make histories isograms, *i.e.,* strings that contain no repeating occurrence of the alphabet. An execution history x defines a total order on its calls: we write $c \prec_x c'$ if the call c precedes the call c' in the execution history x. A replicated execution xs is a mapping from each process to its execution history. The state W of our operational semantics is the pair of the replicated state ss and the replicated execution xs. In the initial state W_0, the state of all processes is the same state σ_0 (which satisfies the invariant $\mathcal{I}$), and their histories are empty.

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

Fig. 4. Replicated State

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

State-conflict. We say that a replicated execution is convergent if all processes store the same state, after all calls are propagated to all processes. Consider a replicated set. As shown in Fig. 6b, if two processes execute an add call c and a remove call c' for the same element with different orders, then their states can diverge. We say that two method calls c_1 and c_2 $\mathcal{S}$-commute, written as $c_1 \diamond_{\mathcal{S}} c_2$, if $c_1 \circ c_2 = c_2 \circ c_1$. Otherwise, they $\mathcal{S}$-conflict, written as $c_1 \bowtie_{\mathcal{S}} c_2$, and need to synchronize with each other. An object is $\mathcal{S}$-commutative if all pairs of calls on it $\mathcal{S}$-commute.

Integrity and Permissibility. The state of the object is expected to maintain its integrity (*i.e.,* satisfy the invariant). For example, the balance of an account is expected to stay non-negative. The body of each method can rely on the invariant in the pre-state. It is then expected to preserve the invariant in its post-state. The notion of *permissibility* requires the invariant to hold in the post-state: we say that a method call c is permissible in a state σ, written as $\mathcal{P}(\sigma, c)$, if $\mathcal{I}(c(\sigma))$. (An impermissible call should not be executed; it should be either rejected, or retried later.) In the execution history of a process, the post-state of a call is the pre-state of the next call. The initial state σ_0 is assumed to satisfy the invariant. Therefore, since every call is permissible in its pre-state, then by induction, every call enjoys integrity in its pre-state. Permissibility leads to integrity. Thus, our next definitions are based on permissibility.

Invariant-sufficiency. There are calls that are always permissible as far as they are applied to a state that has integrity. For example, a *deposit* call never overdrafts the account. Thus, in order to keep them permissible when they are propagated to another process, it is sufficient to execute them on a pre-state that has integrity. We say that a call c is invariant-sufficient if for every state σ, if $\mathcal{I}(\sigma)$ then $\mathcal{P}(\sigma, c)$.

Permissible-Right-Commutativity. However, not all calls are invariant-sufficient. For example, consider an account with balance state b. Consider the execution in Fig. 6b. A *withdraw*$(b/2)$ call c is permissible in process p but is impermissible in process p' where it is executed after a racing *withdraw*(b) call c' that depletes the balance. In the process p', c is pushed to the right of c'. However, if c is *withdraw*$(b/2)$, and c' is *withdraw*$(b/2)$, then c will stay permissible in the other process when it is executed after c'. Similarly, if a *withdraw*(a) call c is permissible with a balance b, then it will stay permissible in the other process when it is executed after any racing *deposit*(a') call c'. A call c $\mathcal{P}$-R-commutes with another c' at a state σ at the pre-state σ, written as $c \triangleright^{\sigma}_{\mathcal{P}} c'$, if at the pre-state σ, permissibility of c holds even after it is pushed right after c'; more precisely, if $\mathcal{P}(\sigma, c)$ then $\mathcal{P}(c'(\sigma), c)$. For example, the *withdraw*$(b/2)$ call $\mathcal{P}$-R-commutes with the *withdraw*$(b/2)$ call at the balance b. A call c $\mathcal{P}$-R-commutes with another c', written as $c \triangleright_{\mathcal{P}} c'$, if it does so for every pre-state σ. A *withdraw*(a) call $\mathcal{P}$-R-commutes with a *deposit*(a') call.

Invariant-conflict. We say that c $\mathcal{I}$-commutes with c' at a pre-state σ, written as $c \triangleright^{\sigma}_{\mathcal{I}} c'$, if c is invariant-sufficient, or if $c \triangleright^{\sigma}_{\mathcal{P}} c'$. For example, a *deposit* call $\mathcal{I}$-commutes with any other call at any balance, and the *withdraw*$(b/2)$ call $\mathcal{I}$-commutes with the *withdraw*$(b/2)$ call at the balance b. Otherwise, c $\mathcal{I}$-conflicts with c' at σ. For example, the *withdraw*$(b/2)$ call $\mathcal{I}$-conflicts with the *withdraw*(b) at the balance b. A call c $\mathcal{I}$-commutes with another c' written as $c \triangleright_{\mathcal{I}} c'$, if it does so for every pre-state σ. For example, a *deposit* call $\mathcal{I}$-commutes with any other call. Otherwise, c $\mathcal{I}$-conflicts with c'. For example, a *withdraw* call $\mathcal{I}$-conflicts with a *withdraw* call.

Conflict. We say that two calls c and c' *commute*, written as $c \blacktriangleleft\blacktriangleright c'$, if they both $\mathcal{S}$-commute and $\mathcal{I}$-commute with each other. Otherwise, we say that they *conflict* written as $c \bowtie c'$. A call is conflict-free if it does not conflict with any other call. For example, a *deposit* call is conflict-free.

Permissible-Left-Commutativity. There are calls that are dependent on their preceding calls to preserve the invariant. For example, consider Fig. 6e. The *withdraw* call c at p' is dependent on the money deposited by a preceding *deposit* call c'; when the *withdraw* call c propagates to process p, it arrives before the *deposit* call c', and it overdrafts. The call c is effectively moved to the left of c'. A call c' $\mathcal{P}$-L-commutes with a call c, written as $c' \triangleleft_{\mathcal{P}} c$, if permissibility of c' holds even if it is moved left before c; more precisely, for every state σ, if c' is permissible in the post-state of the call c on σ, *i.e.*, $\mathcal{P}(c(\sigma), c')$, then c' is permissible in σ, as well, *i.e.*, $\mathcal{P}(\sigma, c')$. For example, a *deposit* call $\mathcal{P}$-L-commutes with a *withdraw* call.

Dependency. A call c' is independent of c, written as $c' \perp\!\!\!\perp c$, if c' is invariant-sufficient, or $c' \triangleleft_{\mathcal{P}} c$. Otherwise, c' is dependent on c, written as $c' \not\perp\!\!\!\perp c$. If c is executed before c' in the issuing process of c', and $c' \not\perp\!\!\!\perp c$, then c' can become impermissible in another process if c is not already executed in that process. For example, a *deposit* call is independent of a *withdraw* call, but a *withdraw* call is dependent on a *deposit* call.

3.2 Semantics

The operational semantics of well-organized replicated data types is presented in Fig. 5. It presents the rules Call to execute an update call locally, Prop to propagate calls to other processes, and Query to execute a query call. We will prove that the semantics guarantees convergence and integrity.

As the label captures, the rule Call accepts an update method call c at the process p, and executes it locally. The call $c = u(v)^r_p$ is decorated with the identifier r and the issuing process p. The rule first checks that the call c is locally permissible $\mathcal{P}(\sigma, c)$ in the current state σ: if c does not preserve the invariant, it is not accepted. The application can either cancel such a call or retry it later. In order to synchronize state-conflicts, the rule then checks a condition called CallSComm. Consider Fig. 6.(a). If the current process p has not executed a call c' that another process p' has executed,

$$\textsc{Call}\ \frac{c = u(v)^r_p \qquad \mathcal{P}(\sigma, c) \qquad \mathsf{CallSComm}(xs, p, c) \qquad xs' = xs\,[p \mapsto (xs(p) ::: c)] \qquad \mathsf{CallIComm}(xs', c) \qquad \sigma' = u(v)(\sigma)}{\langle ss[p \mapsto \sigma], xs\rangle \xrightarrow{\textsc{Call}(p,\ c)} \langle ss[p \mapsto \sigma'], xs'\rangle}$$

$$\textsc{Prop}\ \frac{c = u(v)^r_{p'} \qquad c \in xs(p') \setminus xs(p) \qquad \mathcal{P}(\sigma, c) \qquad \mathsf{PropSComm}(xs, p', p, c) \qquad xs' = xs\,[p \mapsto (xs(p) ::: c)] \qquad \sigma' = u(v)(\sigma)}{\langle ss[p \mapsto \sigma], xs\rangle \xrightarrow{\textsc{Prop}(p,\ c)} \langle ss[p \mapsto \sigma'], xs'\rangle}$$

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

$$\mathsf{CallSComm}(xs, p, c) \ :=\ \forall p', c'.\ c' \in xs(p') \wedge c' \notin xs(p) \rightarrow c \blacktriangleleft\!\!\blacktriangleright_{\mathcal{S}} c'$$

$$\mathsf{PropSComm}(xs, p', p, c) \ :=\ \forall c'.\ c' \prec_{xs(p')} c \wedge c' \notin xs(p) \rightarrow c \blacktriangleleft\!\!\blacktriangleright_{\mathcal{S}} c'$$

$$\mathsf{Pending}(xs) \ :=\ xs \setminus \cap_p\, xs(p)$$

$$\mathsf{CallIComm}(xs, c) \ :=\ \forall c = u(v)_p \in \mathsf{Pending}(xs).\ \mathsf{PRCommAll}(xs, p, c)$$

$$\mathsf{PRCommAll}(xs, p, c) \ :=\ \text{let } x := \mathsf{pre}(xs(p), c),\ \sigma := x(\sigma_0) \text{ in } \forall p'.\ c \notin xs(p') \rightarrow \forall C \subseteq xs \setminus (xs(p') \cup x ::: c).\ \forall c' \in \mathsf{compositions}(C).\ c \blacktriangleright^{\sigma}_{\mathcal{I}} (xs(p') \setminus x) \circ c'$$

Fig. 5. Well-organization semantics. The append operation on lists is written as :::. For histories xs and history x, we lift set operators to histories. For example, $xs \setminus x$ is the set of calls in xs but not in x. For a history x and a call c in x, $\mathsf{pre}(x, c)$ is the prefix of x before c. For a set of calls C, $\mathsf{compositions}(C)$ is the set of all (sequential) compositions of C.

then c and c' should state-commute. Consider when each of the two calls propagate to the other process as shown in Fig. 6.(b). If they state-commute, then the two processes converge; otherwise they can diverge. Therefore, if the two calls state-conflict, the current process p should wait for c' before executing c. Thus, as shown in Fig. 6.(c), when c is propagated to p, the two processes converge. The rule Call also requires the condition CallIComm that we describe below. If the conditions hold, the new call c is appended to the execution history $xs(p)$ of the current process p, and the state $ss(p)$ of p is updated to the result of applying the call $u(v)$ to the current state σ of p.

The rule Prop propagates a call $c = u(v)^r_{p'}$ (from a process p') to the current process p. The call c is in the history of p' but not yet in the history of p. The rule checks a condition called PropSComm. Consider Fig. 6.(d). It checks that if there is a call c' that is executed before c in the other process p', but is not executed in the current process p, then the two calls c and c' should state-commute. Consider Fig. 6.(e), where c' propagates to p after c. If the two calls state-commute, the two processes converge; otherwise, they can diverge. If the two calls state-conflict, the current process p should wait to execute c' before c. As Fig. 6.(f) shows, the two processes execute the two calls in the same order, and converge. The rule further checks that the call c is permissible $\mathcal{P}(\sigma, c)$ in the current state σ. If the conditions hold, the call is locally applied at the current process p.

Let us now consider the condition CallIComm of the Call rule. Consider the execution shown in Fig. 7.(a) where the new call c is being executed. As we saw, the rule Call checks that c is permissible at the issuing process p. The question is whether c will be eventually permissible at p', and delivered there as well. The condition CallIComm leads to this property. Let us see how this condition supports permissibility of c at p'. Consider the calls that p has already executed; in our example, these calls are c_1 and c_2. We move these calls to the left of p' in the same order as p. (By induction, they will be eventually propagated and executed at p'.) We also move the calls that p' has executed but p has not, to the right of p'. In our example, these calls are c'_1 and c'_2. Consider the calls c_1 and c'_1. When c'_1 is executed at p', the call c_1 was already executed at p but not at p'. Therefore, by the state-commutativity conditions above, they can commute in p' without changing

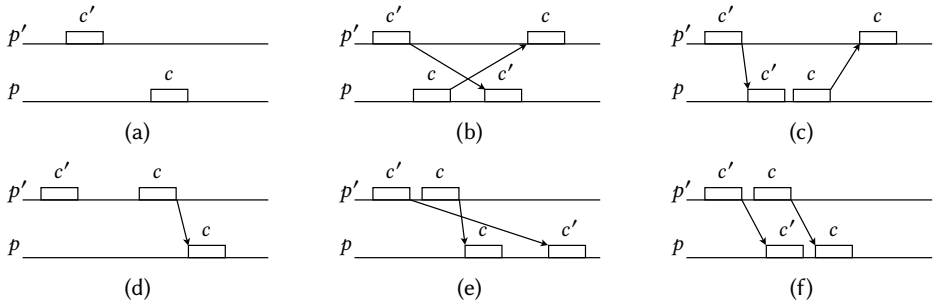


Fig. 6. State-Commutativity

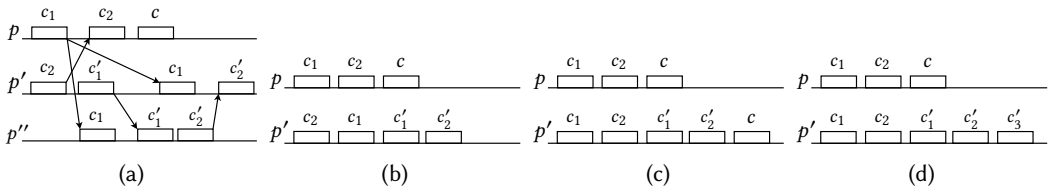


Fig. 7. Invariant-Commutativity

the post-state. The histories resulting from this commute are shown in Fig. 7.(b). (We elide the process p'' and the message passing arrows.) Similarly, the calls c_1 and c_2 have different orders in the two processes p and p'. Therefore, they can commute in p' without changing the post-state. The histories resulting from this commute (and the execution of c at p') are shown in Fig. 7.(c). In this execution, the calls before the new call c in p are c_1 and c_2. These calls now have moved to the left in p', where they have the same order as p. Let σ be the post-state of the composition of c_1 and c_2 at both p and p'. We know that c is permissible at the issuing process p at its pre-state σ. Therefore, c is permissible after the two calls c_1 and c_2 at p' as well. For c to be permissible in p', it should be invariant-commutative with respect to the composition of c'_1 and c'_2. Let the composition of c'_1 and c'_2 be $c' = c'_1 \circ c'_2$. Starting from σ, the call c should be invariant-commutative with c' *i.e.,* $c \triangleright^{\sigma}_{\mathcal{I}} c'$. More generally, when c is issued, let x be the current history xs_p of p. Consider the process p'. The sequence of calls that p' has already executed except calls in x is $xs(p') \setminus x$. Let A be the set of issued calls that are not executed at p' yet. When c arrives at p', the process p' may have further executed any subset of A in any order. Thus, c should be invariant-commutative with $xs(p') \setminus x$, and then any composition c' of any subset C of A. This is the condition PRCommAll for the pending call c itself that CallIComm requires.

CallIComm further requires the condition PRCommAll for the other pending calls. Let us consider how this condition supports permissibility of c at p'. The set of calls A above are already issued when c is being executed at p. What about calls that are issued after c is executed at p, and before c arrives at p'? For example, consider Fig. 7.(d). The call c'_3 is executed at process p' after c is executed at p but before c is propagated from p to p'. The call c should now invariant-commute with $c'_1 \circ c'_2 \circ c'_3$. Consider when the rule Call is executing the call c'_3 at p'. The call c is still pending. The rule Call makes sure that after the call c'_3 is added to p', the condition PRCommAll is maintained for the pending call c. Thus, when the call c propagates to p', it can stay permissible after c'_1, c'_2 and c'_3.

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

Difference with Well-coordination. We next discuss the two differences between well-coordination and well-organization: the local notion of conflict, and not keeping track of dependencies. The former makes well-organization less conservative than well-coordination, and reduces the pairs of methods that are deemed conflicting and need synchronization. The latter reduces the space and time overhead of tracking dependencies. Both differences contribute to lower latency.

Well-coordination required invariant-commutativity globally for all possible states, but well-organization requires it locally *only* for the current state. Well-coordination checks the following invariant-commutativity condition for a new call. Consider a call c that is issued at a process p. If a call c' is executed at another process p', but is not already executed at p, then c should invariant-commute with c', *i.e.,* $\forall c'.\ c' \in xs(p') \wedge c' \notin xs(p) \rightarrow c \triangleright_{\mathcal{I}} c'$. Otherwise, p should wait for c'. Invariant-commutativity requires commutativity from any pre-state: a call c invariant-commutes with c' if either c is invariant-sufficient, or starting from *any* pre-state σ, if c is permissible in σ, then it is permissible in the post-state of executing c' on σ as well. Therefore, invariant-commutativity and -conflict are globally defined on all states. For example, in our AR app use-case in Fig. 2, consider the initial pre-state l. The call two calls $move(a_1)$ and $move(a_2)$ don't $\mathcal{P}$-R-commute with respect to each other at l. Thus, the two calls $\mathcal{I}$-conflict at l. Thus, according to well-coordination, the two calls are considered conflicting globally in the application. However, in order to execute a call c at a process p, well-organization (in the condition PRCommAll) requires c to be invariant-commutative with calls of other processes *starting from only the current state σ of the issuing process p*. Therefore, the notions of invariant-commutativity and -conflict are locally defined for each state. This subtle difference lets well-organization be much less conservative, and avoid synchronizations that are not needed for the current state. In our running example in Fig. 2, consider the pre-state l'. If $move(a_1)$ propagates to process p_2 and moves right to after $move(a_2)$, the ball does not enter restricted zones. The call $move(a_1)$ $\mathcal{P}$-R-commutes with respect to $move(a_2)$ at l'. Similarly, the call $move(a_2)$ $\mathcal{P}$-R-commutes with respect to $move(a_1)$ at l'. Thus, the two calls $\mathcal{I}$-commute at l'. Thus under well-organization, at the pre-state l', processes p_1 and p_2 can execute $move(a_1)$ and $move(a_2)$ concurrently. This example illustrates how, starting from a certain location, a move may only conflict with a subset of other moves. The conflicting moves that can derail the current move into restricted zones can be identified according to the current location.

The second difference is that in contrast to well-organization, well-coordination required a condition called dependency-preservation: when calls are propagated, their dependencies should be preserved. More precisely, consider a call c issued in a process p. Let c' be a call preceding c in p that c depends on. If c is executed at another process p', then c' is expected to have been executed before c in p', *i.e.,* $\forall c'.\ c' \prec_{xs(p)} c \wedge c \not\perp c' \rightarrow c' \in xs(p')$. For example, in our AR use-case, a $move$ call may be dependent on the preceding *move* call to be permissible. For instance, in our running example in Fig. 2, if the pre-state is $\langle 1, 1\rangle$, consider a *move* 4 steps up and then a *move* 2 steps right. The move to the right is permissible but it needs the move up before it; otherwise, the ball can enter the restricted zone. Therefore, well-coordination requires tracking and enforcing these orders. The rule Prop for well-organization that we saw above does not check this condition. Instead, it requires the call c to be permissible in the current state σ of the receiving process p'. (Thus, a call is checked to be permissible not only when the issuing process executes it, but also when it is propagated to another process.) This optimization is based on the following observation. A call can become impermissible if its dependencies are missing; the condition above prevented calls from missing their dependencies. However, there is an easier and less expensive way of ensuring permissibility: directly checking permissibility. Let a process execute a call that it receives only if it is permissible at the current state of that process. If the call is not permissible, buffer it and check again later, similar to calls whose dependencies were not satisfied. By the reliable delivery guarantees, the dependencies will eventually arrive, and make the call permissible. In fact, the call

may become permissible even before its (conservative) dependencies arrive, and can be executed earlier. In our AR use case, well-organization avoids tracking dependencies between *move* calls, and instead queues *move* calls that are received and executes them when they are permissible.

3.3 Guarantees

Every well-organized execution enjoys integrity, convergence and eventual delivery properties. The detailed proofs of the following theorems are in the appendix.

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

Lemma 3.1 (Integrity). *For all ss and p, if* $W_0 \rightarrow^* \langle ss, \_ \rangle$ *then* $\mathcal{I}(ss(p))$.

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

Lemma 3.2 (Convergence). *For all ss, xs, p and p′, if* $W_0 \rightarrow^* \langle ss, xs \rangle$ *and* $xs(p) \sim xs(p')$ *then* $ss(p) = ss(p')$.

As the rule Prop (in Fig. 5) defines, delivering a call c has the pre-condition that c is permissible in the current state σ. Therefore, an immediate question is whether every pending call is eventually delivered. The following theorem answers this question positively: every pending call at any reachable state can be eventually delivered. More precisely, if at a reachable state W, a call c is delivered to p but not p', then there are steps from W that deliver c to p' as well.

Lemma 3.3 (Delivery). *For all* W, τ, p, p' *and* c, *if* $W_0 \xrightarrow{\tau}{}^* W$, $(p, c) \in \tau$, *and* $(p', c) \notin \tau$, *then there exists* τ' *and* W' *such that* $W \xrightarrow{\tau' \cdot (p',\, c)}{}^* W'$.

In fact, the same reasoning as the lemma above can show that every pending call will be eventually delivered in every fair infinite execution. An execution is fair if whenever a rule is enabled, either it eventually executes, or it becomes permanently disabled. A rule is enabled in a state if that state satisfies its pre-conditions. More details are available in the appendix.

4 AR Replication

In this section, we first consider AR applications and their conflicting operations. Then, we present the credit scheme and replication protocol. The protocol is fault-tolerant and can reclaim credits of failed processes. We state and prove that the protocol refines the well-organization semantics that we defined in § 3.2. (Detailed proofs are available in the Appendix, § 9.2.)

4.1 AR Apps

We now define a core AR application and present how its conflicting actions are automatically calculated.

Algorithm 1: AR app for board B and restricted zone R

```
1 class AR-App(B, R)
2     l : Location
3     I(l) := l ∈ B ∧ l ∉ R
4     function move(a : Action, l).
5         return l + a
```

Core App Specification. Alg. 1 captures the specification of a core AR app. An AR app is parametric in terms of the pair $\langle B, R \rangle$ of the board B and the restricted zones R. A virtual object has a location l. The integrity property $\mathcal{I}$ requires the location l of the object to be within the board B, and further not enter the restricted zone R. Users can call the $move(a)$ method to execute an action a that relocates the object. An action a is a pair $\langle d, m \rangle$ where d is the direction and m is the

magnitude of the action. A direction $d \in \mathcal{D}$ is either right X^+, left X^-, up Y^+, down Y^-, inward Z^+, or outward Z^-. The operator + is lifted to vector addition.

Conflicts. We now consider conflicts for the AR app that we just defined. First, we show state-commutativity:

Lemma 4.1. *AR-App is $\mathcal{S}$-commutative.*

This is straightforward from the definition of *move* and the commutativity of vector addition. The *AR-App* class does not experience state-conflicts. Let us now consider invariant-conflicts. As we saw in the Call rule (in Fig. 5), a move should invariant-commute with respect to the moves that other processes have executed but the current process has not. By definition, a call invariant-conflicts with another if it is not invariant-sufficient, and does not permissible-right-commute with respect to the other call. The method *move* is obviously not invariant-sufficient: even if the starting location l is in the allowed areas, a call $move(a)$ can push the object out of the board or into the restricted zone. Therefore, invariant-conflict reduces to permissible-right-conflict. Two *move* calls can permissible-right-conflict with respect to each other. We saw an example in Fig. 2: $move(a_1)$ (which moves right) is permissible at l, but not after it is pushed to the right of $move(a_2)$ (which moves 4 units down) or larger down moves. In fact, the action a_2 is the smallest down action that a_1 conflicts with.

Similar to the example in Fig. 7(c), consider a call $c = move(a)$ at a location l, and two other calls $c'_1 = move(a'_1)$ and $c'_2 = move(a'_2)$ by another process. We want c to permissible-right-commute with respect to c'_1 and c'_2 so that when c is propagated to the other process, c is permissible in that process as well, *i.e.,* $move(a) \triangleright^l_{\mathcal{P}} move(a'_1) \circ move(a'_2)$. What are the actions a_1 and a_2 that fail this condition? We want to find those actions and prevent them.

$$\begin{aligned} &\mathcal{I}(move(a, l)) \wedge \\ &\mathcal{I}(move(a'_1, l)) \wedge \mathcal{I}(move(a'_2, move(a'_1, l))) \wedge \neg\mathcal{I}(move(a, move(a'_2, move(a'_1, l)))) \end{aligned} \tag{1}$$

The condition fails when the action a is permissible at l, and the sequences of actions a'_1 and a'_2 are permissible at l, but then a'_1 and a'_2 make a impermissible. These constraints fall in the theory of integer linear arithmetic. Let $a'_1 = \langle d'_1, m'_1 \rangle$ and $a'_2 = \langle d'_2, m'_2 \rangle$. The implementation uses the Z3 optimizing solver [25] to find the minimum magnitudes m'_1 and m'_2 that make a not permissible-right-commutative.

In the next subsection, we present a protocol that prevents the minimum actions by acquiring enough credits from other processes. This prevents larger conflicting actions in the same direction as well, since the credits required to prevent the minimum actions are more than the credits required to prevent larger actions. Manual calculation of conflicting actions is time-consuming and error-prone. Given an AR environment, a location l and action a, the implementation automatically calculates the function *conflict-actions* that returns the minimum conflicting actions, stores it as a table, and queries the table during execution. It repeats the above calculation for each sequence of actions that can make the call c not permissible-right-commutative, calculating the minimum magnitude for each direction.

4.2 Replication Protocol

In § 4.1, we described how the conflicting actions for each current location and action are calculated. We now present a protocol that is parametric for the conflicting actions.

Conflict-synchronizing Credits. As presented in § 3, in order to preserve the integrity property, conflicting calls should be synchronized. Classical synchronization mechanisms apply total-order broadcast and consensus protocols to preserve the same order for conflicting calls across processes; however, these protocols require multiple rounds of communication. On the other

hand, multi-user AR applications generally require low response times. Our experiments later on show that classical protocols cannot meet these requirements. Therefore, in order to reduce communication, we present a credit mechanism to synchronize between conflicting calls. In brief, calls need to acquire certain amount of credit to execute. Therefore, synchronization of the current call with a remote conflicting call is reduced to acquiring enough credit to inhibit the execution of the other call while the current call is executing. If the process already has the credit required to prevent conflicting calls, it doesn't need to communicate with other processes, saving time. Thus, the credit mechanism facilitates local execution of calls.

Consider the horizontal distance x^+ of a virtual object to a wall (boundary of B) on its right-hand side. The object can move x^+ units to the right; we say that there are x^+ credits to move right. The credits are partitioned and distributed between processes. Further, processes can request and acquire credit from each other. If a process i holds x_i^+ credits, then it can move right up to x_i^+ units locally without communicating with others. Since the total amount of credit in the system is fixed, the process is certain that it will not push the object past the wall even if other processes move the object at the same time. Similarly, a process keeps a credit for each other direction. Each process stores a mapping *holding* from each direction to the amount of credit in that direction that it holds. For example, in a three dimensional board, *holding* is a map from the directions X^+, X^-, Y^+, Y^-, Z^+, and Z^-. We note that credits in opposite directions are dependent: moving in a direction creates credit in the opposite direction. For example, when a process i moves an object right by a magnitude m, it decreases the credit for x_i^+ by m, and increases the credit for x_i^- by m.

Consider a process that wants to execute an action $a = \langle d, m \rangle$ to move the object in the direction d by the magnitude m. If the credit $holding(d)$ that the process holds for the direction d is more than m, it doesn't need to acquire any more credit to do the action. However, it should first make sure that the action a is synchronized with other actions that conflict with it: we say that the action a should be conflict-synchronizing. If the action is conflict-free, no coordination is needed. However, if there are conflicting actions (possibly in orthogonal directions), then those actions can be taken concurrently by other processes, and push the object through the boundaries. Thus, they should be prevented. Consider an action a' that conflicts with a. Before executing the action a, the process acquires enough credit from other processes to leave their credits insufficient to take the action a'. For example, in the Fig. 2 scenario, before moving the object right by a_1 from l, the process has to acquire 4 down credits to prevent other processes from moving the object down past the horizontal boundary of the restricted zone.

The function *conflict-sync-credit* (in Alg. 2 at L. 13) presents the procedure of calculating the amount of credit that should be acquired to prevent conflicting actions. Given a location l and an action a, it returns pairs $\langle d, m \rangle$ of direction d and magnitude m, such that acquiring m credits for direction d prevents conflicting actions. It first retrieves the actions that conflict with a (at L. 49). For example, in Fig. 2, the minimum conflicting action downwards is 4 units at location l. For each such action $\langle d, m \rangle$, it then calculates the amount of credit that can prevent it (at L. 50). The function $bound(d)$ is an upper bound on the number of credits available in the system in the direction d. In our example, the downward bound is 7 units. To prevent the action $\langle d, m \rangle$, less than m credits from the total $bound(l, d)$ should be left out. Thus, the current process needs to have more than the difference of $bound(l, d)$ and m. In our example, $7 - 4 + 1 = 4$ units of downward credit are needed.

Replication Protocol. We now consider the replication protocol. As Alg. 2 shows, the protocol uses reliable broadcast *rb*, and point-to-point links *p2p* as sub-protocols. Fig. 8 shows a running example of the replication protocol that we will use to illustrate its main points, following Fig. 2. The protocol accepts a $Move(a^r)$ request to move the object by the action a. The requests are decorated with unique identifiers r (that can be simply the process identifier and a local request

Algorithm 2: Credit-based Replication Protocol

```
1  sub-protocol:
2      rb : ReliableBroadcast
3      p2p : Point2PointLinks
4  vars:
5      loc : Location                          ▷ Current location
6      active : Identifier = ⊥
7      holding : 𝒟 ↦ ℕ                         ▷ Local credit map
8      kept : Identifier ↦ List[Action]
                                               ▷ Credits kept for an action
9      w : Set[⟨Action, P⟩]                    ▷ Waiting updates
10     acks : Identifier ↦ ℕ
                                               ▷ # of acks received for an action
11     length : 𝒟 ↦ ℕ
12 upon request Move(a^r) where active = ⊥
13     active ← r
14     a ← make-permissible(loc, a)
15     as ← conflict-sync-credit(loc, a)
16     acquire-credit(a + as^r)
17     withdraw-from-holding(a + as)
18     kept ← kept[r ↦ as]
19     loc ← move(a, loc)
20     bound = bound + (−a)
21     acks(r) ← 1
22     rb request broadcast(a^r)
23     active ← ⊥
24     response OK
25 upon rb response deliver(p_s, a^r)
26     if p_s ≠ self then
27         if ¬𝒫(loc, a) then
28             w ← w ∪ {⟨a^r, p_s⟩}
29         else
30             loc ← move(a, loc)
31             bound = bound + (−a)
32             p2p request send(s, Ack(r))
33             foreach ⟨a^r, p_s⟩ ∈ w
34                 if 𝒫(loc, a) then
35                     loc ← move(a, loc)
36                     bound = bound + (−a)
37                     p2p request send(p_s, Ack(r))
38                     w ← w \ {⟨a^r, p_s⟩}
39                     retry foreach
40 upon p2p response deliver(s, Ack(r))
41     acks(r) ← acks(r) + 1
42     if acks(r) = n then
43         ▷ n is # of processes
44         as ← kept(r)
45         deposit-to-holding(as)
46         deposit-to-holding(−a)              ▷ opposite of a
47 function conflict-sync-credit (l : Location, a : Action)
48     as_1 ← conflict-actions(a, l)
           ▷ Retrieve actions as_1 that conflict with a
49     as_2 ← map
50       (λ⟨d, m⟩. ⟨d, bound(d) − m + 1⟩)
51       as_1    ▷ Calculate credits that prevent as_1
52     return as_2
53 function acquire-credit(as^r : List[Action])
54     foreach ⟨d, m⟩ ∈ as
55         if holding(d) < m then
56             m′ ← fraction(m − holding(d))
57             rb request broadcast(Debit(⟨d, m′⟩^r))
58 vars:
59     q                                       ▷ Priority Queue of credit requests
60 upon rb response deliver(p_s, Debit(a^r))
61     q ← q + (⟨p_s, a⟩, r)
           ▷ Add with the identifier r as priority
62 upon periodic
63     foreach ⟨p_s, ⟨d, m⟩⟩^r ∈ q   ▷ priority order
64         if active = ⊥ ∨ active ≺ r then
65             m ← min(m, holding(d))
66             withdraw-from-holding(⟨d, m⟩)
67             p2p request send(p_s, Credit(⟨d, m⟩))
68 upon p2p response deliver(p_s, Credit(a))
69     deposit-to-holding(a)
```

The add operation + is lifted to the list of actions: for $a + as$, if as includes an action with the same direction as a, then the magnitude of a is added to that action; otherwise, a is appended to as.

counter). In the example, at location l, the request is to move 6 units in the X^+ direction. A request $Move(a^r)$ is processed only when processing of another request is not *active* (at L. 13). As we saw in the rule Call of Fig. 5, every call should be permissible at the issuing process. Therefore, as the first step (at L. 15), the requested action a is shrunk so that it does not push the object outside allowed areas. Alternatively, the action could be rejected.

Next, in order to make the action conflict-synchronizing, the process calls the *conflict-sync-credit* function that we considered above (at L. 13) to calculate the credits that it should possess (at L. 16). The result is the pairs as of direction and magnitude. In Fig. 8, *conflict-sync-credit* determines that 4 credits in the Y^- direction are needed. Then, if the process doesn't have enough credit for both

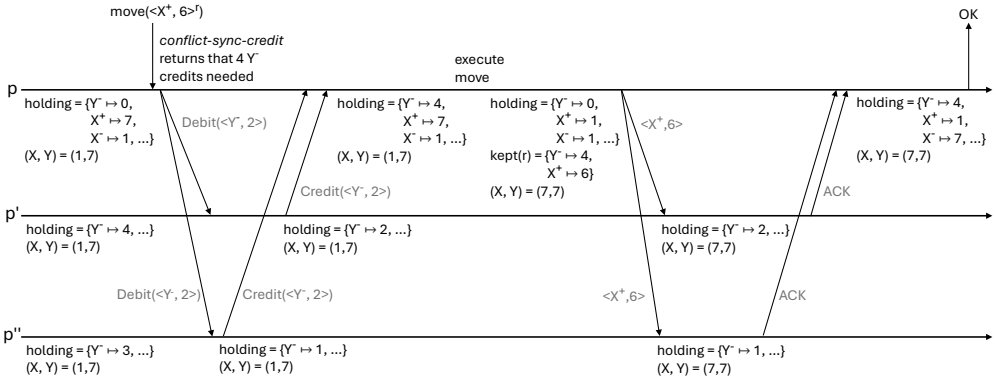


Fig. 8. Run-through of the replication protocol, following the example from Fig. 2.

a and as, it acquires credit from other processes (at L. 17). The function *acquire-credit* (at L. 54) takes a list of actions, and for each action $\langle d, m \rangle$, if the local *holding* has less than the required amount m, it broadcasts a *Debit* request to other processes. Each *Debit* request asks for a *fraction* of the missing number of credits. (The *fraction* function, and further, the number of processes that the request is sent to are configurable.) In the example in Fig. 8, *Debit* requests for 2 Y^- credits each are sent from p to the other processes. When a process receives a *Debit*(a^r) request (at L. 61), it adds a to a priority queue. (The identifier r or other properties of the request can serve as the priority.) Each process periodically traverses its queue in the priority order (at L. 63). It processes a request $\langle d, m \rangle$ with priority r if it doesn't have an active call, or if its active call has lower priority than r. If the process doesn't currently hold the requested number of credits m for the direction d, it contributes the amount that it holds. It withdraws credit from its local *holding*, and sends it in a *Credit* message to the requesting process. In the example, p' and p'' both hold a sufficient number of credits in *holding* to satisfy the request, so they each send 2 Y^- credits to p, and reduce their *holding* accordingly. When a process receives a *Credit* message (at L. 69), it deposits the credit in its *holding*. This is shown in the example where p updates its Y^-*holding* from 0 to 4 credits. If multiple processes request credits for the same direction, the process that is requesting the call with the highest priority gets the credits, and temporarily refuses to release them until it finishes its call.

Next, the process withdraws the credit needed for a and as from its own *holding* (at L. 18). The function *withdraw-from-holding* is atomic, and returns only when enough credit becomes available, and is deducted from the *holding*. The process keeps the credits as in its *kept* map (at L. 19 in Alg. 2), In Fig. 8, p transfers 4 Y^- credits from *holding* to *kept*. It deposits them back later once the action a is executed at all processes. The action a is then executed to update the location *loc* (L. 20), to $(X, Y) = (7, 7)$ at process p in the example. Then, the action a is broadcast to other processes (at L. 23). In the example, a $\langle X^+, 6 \rangle$ action is broadcast from p to the other processes.

As we saw in the rule Prop of Fig. 5, a call received from other processes can be executed only when it becomes permissible. Therefore, when an action a^r from another process is delivered (at L. 26), the process checks that a is permissible in the current location *loc* (at L. 28). If it is not, then the action a together with its sender s are added to the set of waiting calls w (at L. 29). If the action is permissible, the process applies it to the current location, and sends an acknowledgment to the sender (at L. 31-33). This is shown in Fig. 8 where p' and p'' each update the object's position to $(7, 7)$ and send an ACK to p. Later after a is executed at all processes, the sender issues the opposite credit $-a$. Therefore, after sending the acknowledgment, the current process conservatively increases the bound in the opposite direction (at L. 32). The execution of the action can make the waiting actions

permissible. Therefore, the process iterates over its waiting set, and checks if any other action is permissible (at L. 34-35). If such an action is found, it is processed (at L. 36-40).

When a process receives an acknowledgment for a request r (at L. 41), it increments the number of acknowledgments for r in the map *acks*. Once acknowledgments for a are received from all processes, the credits that were held in the *kept* map for conflict-synchronization of a are retrieved and deposited back to the *holding* (at L. 43-46). This is shown in Fig. 8 when p transfers 4 Y^- credits from *kept* back to *holding*. Further, credit in the opposite direction of a with the same magnitude is issued locally (at L. 47). In the example, the X^- credits in *holding* increase from 1 to 7, and the *Move* is complete. We note that issuing this credit before a is executed at all processes compromises the correctness of the protocol.

Fault Tolerance. We described above that each process has a *holding* of credits. If the process fails, those credits may become inaccessible to other processes; thus, some moves may be inhibited. We now extend the protocol to tolerate faults. We start with an example execution with four processes p_a, p_b, p_c and p_d. We consider only the credits in one direction d, say right. As Fig. 9 shows, let (I) the initial holding of each process be 10 for right: $init(p_a) = 10$. (II) The process p_a moves 2 steps right, and then broadcasts the move to others: $moved(p_a) = 2$. It also has exchanges with other processes: (III) it receives 3 credits from p_b, and later (IV) sends 1 credit to p_c, and (V) 1 credit to p_b. Then, (VI) p_d sends 4 credits to p_b. At this point, p_a fails.

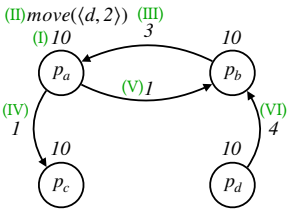


Fig. 9. Example fault tolerance. Arcs show exchanges between processes

How can each other process calculate the credits that p_a had? When a process realizes that p_a failed, it can communicate the exchanges that it had with p_a with other processes. Since p_b sent 3 credits to and received 1 credit from p_a, the net exchange of p_b with p_a was $exchange(p_b, p_a) = 3 - 1 = 2$. The net exchanges of p_c and p_d with p_a were $exchange(p_c, p_a) = -1$ and $exchange(p_d, p_a) = 0$ respectively. The sum of the exchanges is $\Sigma_p\ exchange(p, p_a) = 2 + (-1) + 0 = 1$. Each process knows that the initial value of p_a was 10. It also knows that p_a spent 2 credits for a move, and that it had a sum of 1 exchanges. Thus, each process can calculate that when p_a failed, it had $init(p_a) - moved(p_a) + \Sigma_p\ exchange(p, p_a) = 10 - 2 + 1 = 9$ right credits. There are 3 correct processes; thus, each takes a share of 3 credits. We note that each process can locally calculate the credits to be recovered without expensive synchronization.

What happens if p_b fails too? Then, the processes communicate their exchanges with p_b as well. The net exchange of the two correct processes p_c and p_d with p_b was 0 and 4 respectively. The net exchange of p_c and p_d with p_a was -1 and 0 respectively. The exchanges between failed processes p_a and p_b are not available. We consider the credits that the two failed processes p_a and p_b had together as a whole; therefore, we don't need to consider the individual exchanges between them. The correct processes p_c and p_d had a net exchange of 0 + 4 + (-1) + 0 = 3 with failed processes. The sum of their initial credits was 20, and p_a moved 2 steps. Thus, each process can calculate that p_a and p_b had 20 - 2 + 3 = 21 credits. The 9 credits from p_a were already recovered. Thus, the new credits to be recovered are 21 - 9 = 12. Each correct process takes a share of 12 / 2 = 6 credits.

We now present the fault tolerance protocol to recover the credits that failed processes held. Following a similar argument as the CAP theorem [32], in a partially synchronous network that can be partitioned for an unbounded amount of time, it is impossible to both make the credits of failed processes available, and preserve the bound on the number of credits. In fact, since delays are unbounded, if a heartbeat does not arrive, the process cannot decide whether the other process failed, or the network just partitioned. For co-located AR users that are connected via a local network, the message transmission time can be bounded. This leads to a synchronous network where heartbeat messages can implement a failure detector: if a process fails, it eventually informs every correct

Algorithm 3: Fault Tolerance

```
71 sub-protocol:
72   fd : FailureDetector
73 vars:
74   F                                   ▷ Failed processes
75   ▷ 4 maps from a process p and a direction d:
76   init(p)(d)
77     ▷ initial # of credits that p has for d.
78   moved(p)(d), opposite(p)(d)
79     ▷ # of credits that p spent for move in d.
80     ▷ # of opposite credits that p issued in d.
81   sent(p)(d), received(p)(d)
82     ▷ exchange of self with p for d.
83   exchange(p)(p')(d)
84     ▷ net exchange of p with p' for d.
85   lrc                                 ▷ last recovered credits
86 after executing received action a = ⟨d, m⟩ from
     sender p_s (after L. 31 and L. 36):
87   | moved(p_s)(d) ← moved(p_s)(d) + m
88   | opposite(p_s)(−d) ← opposite(p_s)(−d) + m
                                         ▷ −d is opposite of d
89 after sending credit a = ⟨d, m⟩ to process p_s
     (after L. 68):
90   | sent(p_s)(d) ← sent(p_s)(d) + m
91 after receiving credit a = ⟨d, m⟩ from process
     p_s (after L. 70):
92   | received(p_s)(d) ← received(p_s)(d) + m
93 upon fd response Fail(p)
94   | F ← F ∪ {p}
95   | e ← sent(p) − received(p)
96   | rb request broadcast(Exchange(p, e))
97 upon rb response deliver(p, Exchange(p', e))
98   | r-exchange(p)(p') ← e
99   | C = Π \ F                         ▷ Correct processes
00   | if ∀p ∈ C, p' ∈ F. exchange(p)(p') ≠ ⊥
     |   then
01   |   | wait until Δ is passed since the last fail
02   |   | rc ← Σ_{p'∈F}(init(p')
03   |   |                −moved(p') + opposite(p') +
04   |   |                Σ_{p∈C} exchange(p)(p'))
     |   |                              ▷ recovered credits.
     |   |   ▷ The operator + is overloaded on maps
     |   |   from directions d to # of credits.
05   |   | nrc ← rc − lrc              ▷ new recovered credits
06   |   | lrc ← rc                    ▷ save last recovered credits
07   |   | s ← nrc / |C|               ▷ This process' share
08   |   | holding ← holding + s
```

process, and it informs about the failure of a process only if it has failed. The protocol presented in Alg. 3 uses a failure detector sub-protocol. Each process stores the set of processes that he is informed to have failed $\mathcal{F}$. It also stores several maps: (1) $init(p)(d)$: the initial number of credits that process p has for direction d. Each process is initialized with the *init* map. (2) $moved(p)(d)$: the number of credits that p spent for moves in direction d. (3) $opposite(p)(d)$: the number of opposite credits that p issued in direction d. After a process receives and executes an action $a = \langle d, m\rangle$ from sender process p_s (at L. 31 and L. 36), it increases both $moved(p_s)(d)$ and $opposite(p_s)(-d)$ by m (at L. 87-L. 88). The opposite of direction d is $-d$. (3) $sent(p)(d)$ and $received(p)(d)$: the exchange that the current process had with p for d. After a process sends a credit $a = \langle d, m\rangle$ to a process p_s (at L. 68), it increases $sent(p_s)(d)$ by m (at L. 90). Dually, after a process receives a credit $a = \langle d, m\rangle$ from a process p_s (at L. 70), it increments $received(p_s)(d)$ by m (at L. 92).

When a process is informed of the failure of p (at L. 93), it adds p to the set of failed processes $\mathcal{F}$, and broadcasts the net *exchange* that it has had with p. When a process receives the remote exchange e of a process p with a process p' (at L. 97), it stores it as $r\text{-}exchange(p)(p')$ (at L. 98). The exchanges of correct processes with failed processes are collected once the entries $r\text{-}exchange(p)(p')$ are populated for correct process p and each failed process p' (at L. 100). In order to make sure that reliable broadcast has delivered all the pending messages from failed processes, the process first waits until a period Δ is passed since it was informed of the last failure (at L. 101).[1] It then calculates the credits to be recovered *rc* as the sum of the initial credits of failed processes minus the credits they spent for moves plus the opposite credit they issued plus the sum of collected exchanges (at L. 102). (In the Appendix, Lemma 9.10, we proved that this sum is the amount of credit that is held

[1]To deliver even in a chain of $n-1$ failures, Δ is $(n-1)\times\delta$ where δ is the bounded message transmission time.

by failed processes.) Each process stores the last recovered credits *lrc*. The new recovered credits are the difference of *rc* and *lrc*. Correct processes take equal shares, and add them to their *holding*.

Correctness. In the Appendix, § 9.2, we capture the protocol as a transition system $\Omega_1 \Rightarrow \Omega_2$, and prove that it refines the well-organization semantics $W_0 \rightarrow W$. The immediate result of the refinement is trace inclusion: every trace of the protocol is a trace of the semantics.

Theorem 4.1 (Trace Inclusion). *For all* Ω *and* τ, *if* $\Omega_0 \overset{\tau}{\Rightarrow}^* \Omega$, *then there exists* W *such that* $W_0 \overset{\tau}{\rightarrow}^* W$.

Let us intuitively consider how a refinement holds. A Call step is taken in the $Move(a)$ request handler when the *move* method is locally executed (at L. 20). A Prop step is taken in the $deliver(s, a)$ response handler when the *move* method is remotely executed (at L. 31 and L. 36). We show that these rules are enabled at these lines. For a Call step, permissibility holds directly by the preceding *make-permissible* function call (at L. 15). For a Prop step, permissibility holds directly by the preceding if statements (at L. 28 and L. 35). Further, the protocol trivially satisfies the two conflict-synchronization conditions CallSComm and PropSComm since by Lemma 4.1, *AR-App* is state-commutative.

We now show that the condition CallIComm is satisfied when a Call step is taken: we show that the PRCommAll condition holds firstly, for the call c itself, and then for the pending calls. The protocol executes the call $c = move(a)$ at a process p (at L. 20) only after calculating the conflict-synchronizing credits (at L. 16), and acquiring, withdrawing and keeping them (at L. 17-L. 19). These credits prevent calls that c cannot invariant-commute with respect to. The call c can invariant-commute with any number of calls that can be executed with the remaining credits in any order. Let A be the set of calls that are executed at other processes but not the current process p, and let C be a subset of A. For example, in Fig. 7(a) and (b), the set of calls executed at p' but not p is $C = \{c'_1, c'_2\}$. The call c can invariant-commute with any composition c' of C, *i.e.,* $c \triangleright^{\sigma}_{\mathcal{I}} c'$. Therefore, the condition PRCommAll holds for c.

Further, as we saw in Fig. 7(d) that when a Call step is taken, it should keep the PRCommAll condition for other pending calls as well. When the call c' is executed at the current process p' (at L. 20), consider a pending call c at another process p that is not yet propagated to a process p''. We show by contradiction that the call c invariant-commutes with c'. Otherwise, if c invariant-conflicts with c', then c has acquired enough credit to prevent the execution of c'. A process releases acquired credit (at L. 46) after receiving acknowledgment from all processes (at L. 43). Since p'' has not yet received c (at L. 31 or L. 36), it has not sent an acknowledgment to p (at L. 33 or L. 38). Therefore, p has not released the credits yet. The protocol executes c' at p' only after acquiring and withdrawing credit for c' itself (at L. 17-18). Therefore, the call c' is not executed which contradicts the initial assumption. Thus, the call c invariant-commutes with c'. Thus, when a Call step is taken, the condition PRCommAll is maintained for other pending calls as well.

5 Evaluation

In this section, we detail and evaluate our implementation, which we call Hambazi.

5.1 Implementation

We implemented the Hambazi runtime in Java with the Android framework, and the application was developed based on Google's ARCore SDK CloudAnchor demo app [34]. We used Python Z3 SMT solver API [25] to implement the static calculation in order to find conflicting actions. Specifically, given an AR board and its restricted zones, we use the solver to determine the conflicting actions for the current location l of the virtual object and an action a. As described in § 4, Hambazi inputs the constraints in Eq. 1 to the solver and finds the minimal conflicting actions. It repeats this

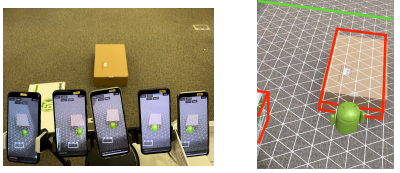
for each location l, and action a in each direction, to generate a JSON table of conflicts that it queries at runtime. Our experiments showed that naively generating the table by calling the solver exhaustively for all possible locations and combinations of other actions is time-consuming. To accelerate the calculation of the table, Hambazi makes several optimizations. First, the actions of a potentially conflicting sequence might be infeasible due to the bounded nature of other requests. With the knowledge of human finger tapping frequency [86], the number of users, and the magnitude of their actions, we can bound the possible actions of other users. Therefore, other users' aggregated actions may not be large enough to cause a conflict, so we can avoid acquiring credit for such infeasible conflicting sequences. Second, for simple topologies (*e.g.,* a single restricted zone), we notice that conflicting actions follow simple algebraic rules. Thus, we compute these actions directly from these rules instead of calling the solver. As a sanity check, we compared the solver's results to the rules' results, and confirmed that they matched perfectly. For more complex topologies (*e.g.,* with multiple restricted zones), we use the algebraic rules wherever possible, and the solver only in intermediate regions that are in the vicinity of more than one restricted zone. More details of these calculations are available in the Appendix (§ 10).

5.2 Experimental Results

We evaluate Hambazi, seeking to answer the following questions:

- How is AR user experience in terms of latency, and how recent is her local view in terms of the virtual object's position? (§ 5.3)
- How well does the system handle an increasing volume of user actions (§ 5.4), and increasing number of users (§ 5.5)?
- How do different topologies (*e.g.,* with more, irregular or moving restricted zones) impact the amount of coordination needed and hence system performance? (§ 5.6)
- If users have different network latencies, do they experience fairness in terms of how long it takes their actions to be viewed by other users? (§ 5.7)
- What is the impact of failures on the system in terms of latency and throughput? (§ 5.8)

Setup. As Fig. 10 shows, the experiments were conducted with up to 7 Android smartphones (a mix of Google Pixel 4, 5, and Samsung S10 devices) as replicas in a set of 3D AR game boards with restricted zones shown in Fig. 14b, including one with a moving restricted zone. During a 5-minute trial, each device continuously calls $move(a)$ to modify the same virtual object's position (the green character in Fig. 10b) with an action a in random direction (up, down, left, right, inward, outward) and magnitude (uniformly between [0%, 5%] of the total board width and length), with call frequency following an exponential distribution with a mean inter-arrival time of 140 milliseconds by default. 140 ms was chosen because the finger tapping frequency of humans is around 5.62 Hz (178 ms) [86]. The credits are initially distributed equally. Credit requests for 100% of the needed credits are sent to $(N + 1)/2$ of the devices (these parameters are set based on our empirical evaluations that are available in the Appendix, § 10). Each experiment was performed 3 times. Devices communicate over WiFi with TCP connections. The experiments were conducted on a home WiFi network with an average download (upload) bandwidth of 340 (10) Mbps, and an average round trip time (RTT) of 187 ms with a standard deviation of 143 ms between a pair of devices.



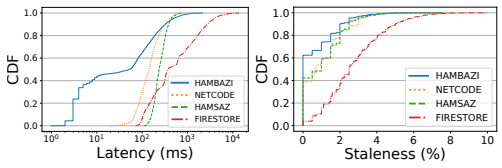
(a) Experiment setup (b) Device screenshot

Fig. 10. Example setup with 5 devices, manipulating the same virtual object (green character). Restricted zones are outlined in red, and AR board boundaries are outlined in green.

Baselines. We compare Hambazi with three baselines. (1) Hamsaz (well-coordination [40]) uses total-order-broadcast to deliver conflicting calls (such as AR *move* calls) in the same order to all processes. To run Hamsaz on Android, we modified MicroRaft [30], an open-source implementation of Raft [72] with minimal dependencies on external libraries. The expected latency is at least 2 RTT between devices (one RTT for the device that issues a call to send it to the leader, and the leader to broadcast it to all devices, plus one RTT for the leader to gather the majority responses and then broadcast the result to all devices). (2) Netcode is a set of principles used by major game engines, such as Unity, that rely on a host to receive and reconcile client actions [31, 90]. We adopted client-hosted network architecture, where one client acts as the host, and call results are immediately displayed and possibly rolled back later (reconciled) if conflicts are discovered later by the host. The latency for a call to be confirmed with the host is at least 1 RTT between devices. (3) Finally, Firestore is a server-based approach that relies on black box transaction services provided by the Google Cloud Firestore database [35]. This baseline is the default implementation in Google's AR demo apps. To preserve the integrity property, calls are sent to Firestore as atomic transactions. A call succeeds in overwriting the object's position in the database only if no other device has updated the position in the interim. Thus the expected latency for each call is at least 1 RTT between the Firestore server and the device, plus the server-side processing delay.

Evaluation metrics. We evaluate Hambazi in terms of several measures of AR user experience. The *latency* of a call measures the time spent between when a device submits a call to move the virtual object until the movement (which respects integrity) is confirmed and rendered on display. The *location staleness* measures the distance between virtual object's local position and an "oracle" view, and is represented as a percentage of the AR board length. By "oracle" view, we refer to a hypothetical system that instantaneously accepts and propagates all devices' calls in the order they are issued. Finally, we evaluate the *throughput* of the system, indicating the rate with which the system can successfully process calls (without dropping them). The full-sized plots of all results in subsequent subsections and additional 2D AR game boards results are in the Appendix (§ 11).

5.3 Latency, Location Staleness, and Throughput



(a) Latency (ms) (b) Staleness (%)

Fig. 11. Compared to baselines, Hambazi has lower latency 90% of the time, and the lowest location staleness. (Full-sized plots are in the Appendix Fig. 24.)

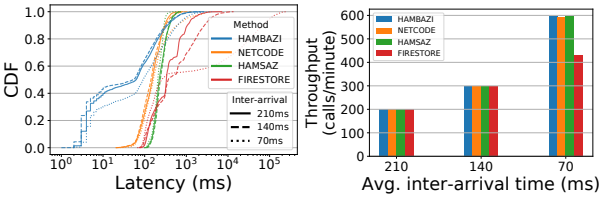
We first evaluate the key metrics of latency, location staleness, and throughput. We focus on the "Middle3D" board topology (Fig. 14b); results for other boards are similar and are available in the Appendix, Fig. 31-33. Fig. 11a shows the cumulative distribution function (CDF) of the latency. Hambazi (with an average/median latency of 119.8/53.0 ms) outperforms the Netcode and Hamsaz baselines (with average/median latency of 172.3/142.0 ms and 260.7/237.0 ms, respectively) 90% of the time, which is a reduction of the average latency by 30.5% and 54.1% respectively. Hambazi avoids communication by performing local actions as often as possible (46.4% of the actions), resulting in nearly zero latency for those local actions. Note that for Netcode, the latency is for the result to be confirmed by the host. Alternatively, actions could be immediately applied resulting in a lower display latency (9.7 ms) but such actions may violate the integrity property and expose rollbacks to AR users. For the Firestore baseline, due to transaction contention on the server, the latency is significantly higher, with an average/median latency of 1033.5/369.0 ms and a long tail of up to 13 seconds. For location staleness in Fig. 11b. Hambazi (with average/median location staleness of 0.67%/0.0% of the board) keeps the users' local views best synchronized. Compared to Netcode and Hamsaz, Hambazi reduces location staleness by an average of 36.8% and 35.6%

respectively. In terms of absolute numbers, 1.0% corresponds to a physical distance of 2 cm, which is within the error range currently seen in off-the-shelf server-based AR [78].

In terms of system throughput, Hambazi successfully completes all calls while Netcode, Hamsaz, and Firestore baselines tend to drop a small number of calls, with 0.3%, 0.3%, and 0.6% unfinished calls per minute respectively. (The plots are available in the Appendix, Fig. 24c.) We also experimentally verified that the integrity and convergence properties are satisfied except for Netcode, when it is allowed to roll back.

5.4 Impact of Request Load

Next, we study the impact of the request load by varying the mean inter-arrival call time from 70 ms (intense load) to 140 ms (moderate load), to 210 ms (light load). As Fig. 12a shows, Hambazi outperforms Hamsaz and Netcode 72% and 70% of the time, respectively, with Hambazi having a median latency of 92.0 ms vs. 206.0 ms and 161.0 ms for the two baselines under the intense load. Firestore has difficulty handling the contention and has a median latency of 257.0 ms. This is because Firestore's ordering of conflicting action calls causes a sequential wait for the older actions to be finished. Increasing load impacts Hambazi minimally with a slightly longer tail up to around 1700 ms under the heaviest load.



(a) Latency (b) System throughput

Fig. 12. Varying the mean inter-arrival times from light load (210 ms) to intense load (70 ms). Even with the intense load, Hambazi yields the lowest latency 70% of the time and finishes all calls. (Full-sized plot is available in the Appendix, Fig. 26.)

In Fig. 12b, even with the intense load, Hambazi outperforms the system throughput of all the baselines. Hambazi can maintain the service and finish all calls, while Hamsaz and Netcode cannot complete 0.2% of the calls. Even with the higher loads, Hambazi can finish almost all calls since pending actions can be batch-processed as other actions are issued. Also, due to the calls being issued continuously (based on the inter-arrival time), there is idle time in between for the system to process the pending calls. However, Firestore fails to complete 27.8% of the requested calls, due to the high read-then-modify contention on the server. For location staleness, even with the intense load, Hambazi outperforms all the baselines. (The plot is available in the Appendix, Fig. 26b.)

5.5 Scalability

In this subsection, we examine the scalability of Hambazi as more devices join the AR session and issue calls. In Fig. 13, we plot the median latency for different numbers of users (1 to 7) participating in the session. (Note that Hamsaz requires at least two devices in order to perform the leader election, as does Netcode for the host-client architecture.) The results show that Hambazi's latency remains stable, with a median latency of under 135.0 ms even with 7 devices. For the case of 1 device, all the actions are conflict-free and have a median latency of 8.0 ms for Hambazi, corresponding purely to computation and not communication time. In contrast, with 2 or more devices, Netcode requires 1 RTT and has a median latency between 137 to 160 ms while Hamsaz requires at least 2 RTT to commit an action, resulting in a median latency between 207 to 280 ms. The latency for Hamsaz slightly decreases from 2 to 3 devices because the Hamsaz leader can process calls in batches if it receives

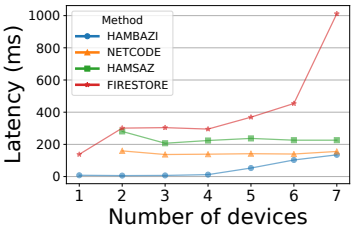


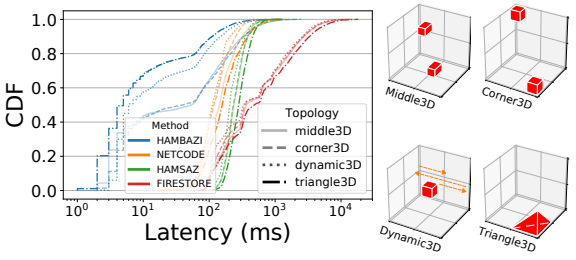
Fig. 13. With an increase of devices issuing calls, Hambazi still benefits from conflict-free actions and results in the lowest latency. (Full-sized plot is available in the Appendix, Fig. 28.)

them while waiting for a majority to respond to the previous call. Therefore, for a fixed inter-arrival time, when more users issue calls, more calls can be batched, yielding slightly lower latency. However, these benefits subside as the number of users grows to 7.

Our experiments also show that as the number of devices increases, Hambazi and Hamsaz slightly increase location staleness, but keep it under 1.0%, and Hambazi is more accurate. On the other hand, Netcode increases up to 1.5% as users make tentative local moves. (The plots for location staleness and throughput are available in the Appendix, Fig. 28b.)

5.6 Impact of Board Topology

We examine the impact of different board topologies on performance, as some topologies may require more coordination, impacting performance. We experiment with four sample 3D AR boards (200 cm x 200 cm x 200 cm) with different shapes of the restricted zones, shown in Fig. 14b. Boards with more space between restricted zones tend to be more difficult because movements in that space often require acquiring credit in orthogonal directions. We also include one dynamic board where the restricted zone moves back and forth horizontally at a constant speed (5 cm/s), simulating the scenario where a virtual object should not intersect with a moving object/person. To account for a moving zone, Hambazi predicts the future location of the zone, and acquires credits accordingly. Since the time to acquire credits may vary, it reconsiders the location of the zone right before taking action. Similar results with five additional 2D topologies are presented in the Appendix, Fig. 33, 34.



(a) Latency (b) Boards

Fig. 14. Board topology affects the amount of conflict-free actions, but even for more challenging boards, Hambazi outperforms the latency of the baselines. (Full-sized figure is available in the Appendix, Fig. 30.)

As Fig. 14a shows, Hambazi improves latency even for challenging topologies since it avoids communication for conflict-free actions. The Middle3D topology, which benefits from 46.4% conflict-free actions, yields a median latency of 53.0 ms. The Triangle3D topology has the most conflict-free actions among all topologies, at 75.8%, and is the easiest topology; therefore, it has a median latency of 5.0 ms. Since Netcode, Hamsaz and Firestore do not benefit from conflict-free actions, the topology difficulty does not impact them, yielding a median latencies 86.0, 143.0, and 170.0 ms, respectively, which are higher than Hambazi's. Considering location staleness, more difficult boards generally have higher location staleness, across the three methods. However, the location staleness of all topologies are all under 4% with only small differences between board topologies. The throughput of all three methods for different board topologies behaves similarly, with Hambazi finishing all calls while Netcode and Hamsaz have 0.3-0.4% and Firestore has 0.2-0.8% unfinished calls. (The plots are in the Appendix, Fig. 30-31.)

Table 1. The time to generate the table with Z3 SMT solver and algebraic rules.

Topology (Fig. 14b)	Optimized time
Middle3D	6 hrs
Corner3D	6 hrs
Dynamic3D (1 frame)	2 hrs
Triangle3D	26 mins

Finally, we report on the offline computation time for our solver optimizations (§ 5.1) to generate the conflict table for each board topology. Although we calculate accurate conflicts for each location, we note that precision can be traded for speed by conservatively calculating conflicts for subspaces rather than individual locations. This table is computed statically only once, and is then queried repeatedly at runtime, alongside the algebraic rules described in § 5.1.

5.7 Impact of Network Latency

We seek to understand the impact of slower networks on action latency, both when devices have homogeneous and heterogeneous network latencies. To systematically evaluate the impact of networks, we simulate controlled network environments. The simulator uses the same Java code as the Android app, running each client as a separate process on a local machine, and replacing WiFi communications with an emulated link to match the RTT statistics from the previous experiments, which had a mean RTT of 20 ms with a standard deviation of 100 ms labeled as "1x RTT" in the following experiments.

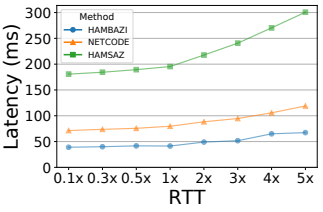


Fig. 15. Slower networks impact baselines, while Hambazi achieves a median latency of 67.3 ms. (Full-sized plot is available in the Appendix, Fig. 35.)

Homogeneous network latencies. Fig. 15 shows the impact of increasing the mean RTT from 0.1x up to 5x with the median latency. The results show that Hambazi still benefits from the conflict-free actions and is less impacted by network RTT, achieving a median latency of 67.3 ms even with 5x RTT. In contrast, Netcode and Hamsaz have greater reliance on the network, and their median latency grows as the network RTT increases.

Heterogeneous network latencies. We seek to understand the fairness of the latencies experienced by the devices when one of the devices has poor network conditions. In this experiment, Device 0 has an average RTT of 200 ms (corresponding to the 10x RTT setting) while the others remain at 20 ms (corresponding to the 1x RTT setting). In Fig. 16, we present the median latency of performing an action from each device's perspective. In Hambazi, all devices have around 50 ms median latency. Hambazi benefits from conflict-free actions to avoid communication, and thus, is less affected by heterogeneous RTT. In contrast, Netcode and Hamsaz have a substantially higher median latency, with a greater disparity between Device 0 and the other devices, indicating greater unfairness. Note that Device 2 is the leader/host device in the Hamsaz/Netcode. Therefore, Device 2 has a lower median latency with Hamsaz and display latency with Netcode.

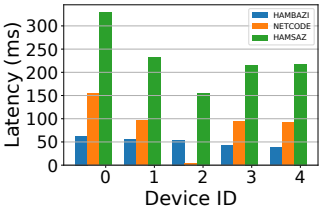
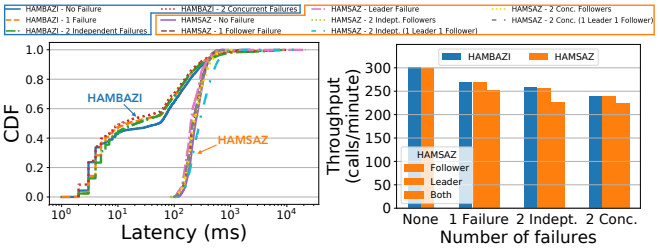


Fig. 16. Device0 has slower networks. Hambazi is less impacted by network heterogeneity, while Netcode and Hamsaz have up to 1.6x and 1.5x of the latency for Device0 compared to other devices. (Full-sized plot is in the Appendix, Fig. 37.)

5.8 Fault Tolerance

Finally, we evaluate the impact of device failures on Hambazi compared to Hamsaz. (Netcode does not have a built-in failure recovery mechanism.) Both Hambazi and Hamsaz trigger a failure detector every 10 seconds. The configurable wait period Δ of Alg. 3 is set to 4 seconds. During the 5-minute trial in Middle3D topology, we suspend different combinations of devices to simulate failure. Specifically, we suspended 1 or 2 devices midway through a trial, denoted as "1 Failure" and "2 Concurrent Failures". We suspended 1 device midway



(a) Latency (b) Throughput

Fig. 17. Hambazi continues operations in the presence of failures. The overall latency is not affected, with slightly longer tails depending on the pre-configured recovery wait time. Throughput decreases because failed devices cannot contribute calls. (Full-sized plots are in the Appendix, Fig. 39.)

and the second device 90 seconds later, denoted as "2 Independent Failures" in Fig. 17. For comparison, we suspended either a leader or follower device while running Hamsaz, marked as "Leader Failure" and "1 Follower Failure". Additionally, we suspended two devices simultaneously: either two followers ("2 Conc. Followers") or the leader and a follower ("2 Conc. (1 Leader 1 Follower)"). We also tested the same setup for two independent failures, denoted as "2 Indept. Followers" and "2 Indept. (1 Leader 1 Follower)".

As Fig. 17a shows, Hambazi's latency is generally not greatly impacted by failures, because it does not pause system operation when failures are present, and devices can still perform local actions during recovery. Further, since failures reduce the total number of devices present, the remaining devices have more credits on hand, and gathering credits becomes easier. Therefore the fraction of local actions increases, *e.g.,* from 46.6% (no failures) to 54.4% (2 concurrent failures), resulting in slightly improved latency in Fig. 17a. However, the presence of failures results in a longer tail of up to 11 seconds, due to conflicting actions that are issued when failures occur and cannot be executed before failures are detected and recovered. For example, the maximum latency is 2141.0 ms in the no failure case, versus a higher maximum of 11.13 s with 2 concurrent failures. The throughput of Hambazi in Fig. 17b decreases with more failures, because failed devices cannot process calls. (Results for other boards are similar and available in Appendix, Fig. 40-Fig. 47.)

In contrast, when the leader device in Hamsaz fails, the system becomes unavailable for at least 3 seconds until a new leader is re-elected. Requested actions cannot be processed during this interval, resulting in a decreased throughput in Fig. 17b. Similarly, suspending a follower impacts the throughput since the follower is unable to contribute calls to the system. Regarding latency, Fig. 17a captures the latency of the calls processed when a correct leader is present, and thus it remains 2 RTT even with failure cases.

6 Related Work

Coordination in AR and Games. Approaches in practice typically involve centralized coordination in the cloud and optimistic concurrency models. Popular game engines like Unity [90] follow netcode principles [31, 91], where clients make local actions (*e.g.,* updating hit points, player movements) and roll back if conflicts are discovered later at the central server. However, such rollbacks can be detrimental to user experience. For AR, Microsoft Hololens [63] and Google ARCore/Firebase [34, 35] follow optimistic concurrency models, where the first write wins, and other replicas must refresh their state before re-trying a write. However, such approaches are insufficient because AR application cannot tolerate high latencies from communication with the cloud or contention in optimistic concurrency models. In contrast, Hambazi uses a peer-to-peer coordination-avoiding protocol with formal guarantees of convergence and integrity.

Several works avoid cloud communication and propose peer-to-peer systems for massively multiplayer online games [11, 12, 43]. These works focus on system design, and interest management techniques that only propagate objects/player updates to relevant replicas. Our protocol coordinates the updates within a given interest set defined by such works. Nomad [37] proposes consensus for latency-sensitive applications but conservatively requires total ordering, and adjusts the leader in a Paxos-like approach, and thus cannot enjoy speedups from local operations as in Hambazi. DyConits [27] takes inspiration from TACT [98] and designs centralized coordination protocols for virtual environments, but only handles synchronization after all calls are accepted. This is unlike our work that determines and prevents conflicting calls. Finally, none of the above works account for AR-specific properties such as its restricted play areas.

Replicated Data Types. Consensus and total-order broadcast protocols [24, 28, 48, 70, 72, 73] provide strong consistency [38, 74]. However, practitioners [23, 47, 75, 92] soon realized that they do not provide the required low latency [2, 14, 15] for industrial applications, and opted for relaxed

notations of consistency. On the other hand, the large collection of relaxed consistency notions [83] can be more efficiently provided [49, 76]. However, the safety of replication on top of these notions is more subtle. Convergent and Commutative Replicated Data Types (CRDTs) [84, 85] and similar notions [3, 46, 80] formally define replicated data types that eventually converge. Followup works define specifications for the functional correctness of these types [56, 93], their composition [22, 61, 94], and verification [13, 18, 29, 33, 59, 65, 66, 68, 69, 88, 99, 100]. They were later followed by more expressive convergent types such as cloud, mergeable, and reactive types [16, 17, 42, 42, 64]. In this paper, we proved that well-organization guarantees convergence as well.

In addition to convergence, integrity is an important required property. Certain operations [5, 67] can preserve integrity under relaxed consistency, while others [6] need strong consistency. Therefore, several hybrid models including IPA [7], Sieve [52–54], Indigo [8, 9], CISE [36], Quelea [87], Carol [51], Hamsaz (well-coordination) [40, 41, 55], ECRO [26] and ConSysT [44] consider the semantics of each operation to execute it under either relaxed or strong consistency. Well-organization that this paper presents is inspired by well-coordination with two important differences (described in § 3) that are crucial to support more local executions, and lower latency. In particular, the above models define the notion of conflict conservatively for all states. In contrast, well-organization is based on a local notion of conflict for the current state. Further, this paper presents a credit-based protocol to implement well-organization that further supports local execution of even conflicting operations.

Escrow and reservation mechanisms [10, 39, 45, 71, 77, 79, 97] split resources between processes that each can locally consume without violating integrity. Similarly, our protocol uses credits to enforce limits for each spatial direction. However, not entering the restricted zones is an integrity property on the values of multiple dimensions. Multiple separate bounded counters from previous works would be insufficient to enforce it: concurrent moves in different directions can misplace the virtual objects, as illustrated through the example in § 2. This paper presents a protocol that considers conflicting actions across counters (directions), and acquires credit from other counters to prevent conflicts. Escrow has been generalized to logical assertions and locks [8, 57, 60, 81, 95, 96] to accelerate storage systems. Each process preserves a predicate that restricts its state to a subspace; if a process finds that it cannot execute an operation without violating its predicate, it performs a global synchronization across processes and installs new predicates. This paper avoids global synchronization: for a given operation, a process acquires enough credit such that other processes are unable to violate integrity. Finally, in contrast to this paper, previous works did not consider fault tolerance and recovery of escrows, or assumed that the hosting data centers [8, 81] maintain the stability of each replica.

7 Conclusions

This paper posed the new problem of multi-user spatial coordination for AR capturing their integrity, convergence, and latency requirements. In particular, a virtual object should respect spatial boundaries. The paper introduced well-organized replicated data types to meet these requirements, supporting low latency through a local notion of conflict. The paper then develops a credit scheme and replication protocol that formally implements well-organized virtual objects. It further presents a tool that, given an AR environment, applies constraint solvers to automatically derive conflicts, and instantiates the replication protocol to synthesize custom correct-by-construction coordination. Empirical evaluation on Android devices demonstrates significant improvements in latency, staleness, throughput, and scalability compared to baselines from the literature and from practice.

Acknowledgments

This work was partially funded by NSF CAREER 1942700, NSF CAREER 2437238, and DARPA YFA D22AP00146-00.

Data-Availability Statement

All the code and data produced in this paper are available on Zenodo https://zenodo.org/records/14941458.

References

[1] 2025. Cubism. https://www.cubism-vr.com/.
[2] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design. *Computer* 45, 2 (2012), 6 pages. https://doi.org/10.1109/MC.2012.33
[3] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency analysis in Bloom: A CALM and collected approach. In *Conference on Innovative Data Systems Research*. 249–260.
[4] Kittipat Apicharttrisorn, Bharath Balasubramanian, Jiasi Chen, Rajarajan Sivaraj, Yi-Zhen Tsai, Rittwik Jana, Srikanth Krishnamurthy, Tuyen Tran, and Yu Zhou. 2020. Characterization of multi-user augmented reality over cellular networks. In *IEEE SECON*. https://doi.org/10.1109/SECON48991.2020.9158434
[5] 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
[6] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2015. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1327–1342. https://doi.org/10.1145/2723372.2737784
[7] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. [n. d.]. IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases. *Proceedings of the VLDB Endowment* 12, 4 ([n. d.]). https://doi.org/10.14778/3297753.3297760
[8] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting Consistency Back into Eventual Consistency. In *European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. ACM, New York, NY, USA, Article 6, 16 pages. https://doi.org/10.1145/2741948.2741972
[9] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. 2015. Towards Fast Invariant Preservation in Geo-replicated Systems. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 121–125. https://doi.org/10.1145/2723872.2723889
[10] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. 2015. Extending eventually consistent cloud databases for enforcing numeric invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 31–36. https://doi.org/10.1109/SRDS.2015.32
[11] Ashwin Bharambe, John R Douceur, Jacob R Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. 2008. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. *ACM SIGCOMM Computer Communication Review* 38, 4 (2008), 389–400. https://doi.org/10.1145/1402958.1403002
[12] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. 2006. Colyseus: A Distributed Architecture for Online Multiplayer Games. In *NSDI*, Vol. 6. 12–12.
[13] A. Bouajjani, C. Enea, and J. Hamza. 2014. Verifying Eventual Consistency of Optimistic Replication Systems. In *Proc. POPL*. https://doi.org/10.1145/2535838.2535877
[14] Eric Brewer. 2012. CAP twelve years later: How the" rules" have changed. *Computer* 45, 2 (2012), 23–29. https://doi.org/10.1109/MC.2012.37
[15] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7. Portland, OR, 343477–343502. https://doi.org/10.1145/343477.343502
[16] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 691–707. https://doi.org/10.1145/1869459.1869515
[17] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. 2012. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*. Springer, 283–307. https://doi.org/10.1007/978-3-642-31057-7_14
[18] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proc. POPL*. https://doi.org/10.1145/2578855.2535848
[19] CBS News. 2018. New augmented reality technology could help firefighters save lives. https://www.cbsnews.com/news/c-thru-new-augmented-reality-technology-would-aid-firefighters/.

[20] Sarah E Chasins, Elena L Glassman, and Joshua Sunshine. 2021. PL and HCI: Better together. *Commun. ACM* 64, 8 (2021), 98–106. https://doi.org/10.1145/3469279

[21] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, et al. 2017. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *ACM/IEEE Symposium on Edge Computing*. https://doi.org/10.1145/3132211.3134458

[22] Kevin Clancy and Heather Miller. 2017. Monotonicity Types for Distributed Dataflow. In *Proceedings of the Programming Models and Languages for Distributed Computing*. ACM, 2. https://doi.org/10.1145/3166089.3166090

[23] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288. https://doi.org/10.14778/1454159.1454167

[24] 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/10.1145/2463676.2465288

[25] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[26] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: building global scale systems from sequential code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485484

[27] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. 2021. Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 126–137. https://doi.org/10.1109/ICDCS51616.2021.00021

[28] Cezara Drăgoi, Josef Widder, and Damien Zufferey. 2020. Programming at the edge of synchrony. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428281

[29] Michael Emmi and Constantin Enea. 2018. Monitoring Weak Consistency. In *Proc. CAV*. https://doi.org/10.1007/978-3-319-96145-3_26

[30] Kahveci Ensar. 2021. MicroRaft. https://microraft.io/.

[31] Gabriel Gambetta. 2022. Fast-Paced Multiplayer. https://www.gabrielgambetta.com/client-server-game-architecture.html.

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

[33] Victor BF Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28. https://doi.org/10.1145/3133933

[34] Google. 2023. Cloud Anchors allow different users to share AR experiences. https://developers.google.com/ar/develop/cloud-anchors.

[35] Google. 2023. Transaction serializability and isolation. https://firebase.google.com/docs/firestore/transaction-data-contention.

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

[37] Zijiang Hao, Shanhe Yi, and Qun Li. 2019. Nomad: An efficient consensus approach for latency-sensitive edge-cloud applications. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2539–2547. https://doi.org/10.1109/INFOCOM.2019.8737658

[38] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[39] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. ACM, New York, NY, USA, 279–293. https://doi.org/10.1145/2987550.2987559

[40] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. In *Proceedings of Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3290387

[41] Farzin Houshmand, Javad Saberlatibari, and Mohsen Lesani. 2022. Hamband: RDMA replicated data types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 348–363. https://doi.org/10.1145/3519939.3523426

[42] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29. https://doi.org/10.1145/3360580

[43] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. 2004. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM 2004*, Vol. 1. IEEE. https://doi.org/10.1109/INFCOM.2004.1354485

[44] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. 2020. Rethinking safe consistency in distributed object-oriented programming. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428256

[45] Narayanan Krishnakumar and Arthur J Bernstein. 1992. High throughput escrow algorithms for replicated databases. In *VLDB*, Vol. 1992. 175–186.

[46] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M Hellerstein. 2022. Keep CALM and CRDT On. *Proceedings of the VLDB Endowment* 16, 4 (2022), 856–863. https://doi.org/10.14778/3574245.3574268

[47] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922

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

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

[50] Kiron Lebeck, Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. 2017. Securing Augmented Reality Output. In *IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2017.13

[51] Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černỳ. 2019. Sequential programming for replicated data stores. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28. https://doi.org/10.1145/3341710

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

[53] 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. https://doi.org/10.1145/2745947.2745955

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

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

[56] Hongjin Liang and Xinyu Feng. 2021. Abstraction for conflict-free replicated data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 636–650. https://doi.org/10.1145/3453483.3454067

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

[58] Shengmei Liu, Xiaokun Xu, and Mark Claypool. 2022. A survey and taxonomy of latency compensation techniques for network computer games. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–34. https://doi.org/10.1145/3519023

[59] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. 2020. Verifying replicated data types with typeclass refinements in Liquid Haskell. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. https://doi.org/10.1145/3428284

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

[61] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. 184–195. https://doi.org/10.1145/2790449.2790525

[62] Meta. 2022. Guardian System - Oculus Developer Center. https://developer.oculus.com/documentation/native/pc/dg-guardian-system/.

[63] Microsoft. 2022. How to create and locate anchors using Azure Spatial Anchors in Unity. https://learn.microsoft.com/en-us/azure/spatial-anchors/how-tos/create-locate-anchors-unity.

[64] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. 2019. A fault-tolerant programming model for distributed interactive applications. *Proceedings of the ACM on Programming Languages* OOPSLA (2019), 1–29. https://doi.org/10.1145/3360570

[65] Kartik Nagar and Suresh Jagannathan. 2019. Automated parameterized verification of crdts. In *International Conference on Computer Aided Verification*. Springer, 459–477. https://doi.org/10.1007/978-3-030-25543-5_26

[66] Kartik Nagar, Prasita Mukherjee, and Suresh Jagannathan. 2020. Semantics, Specification, and Bounded Verification of Concurrent Libraries in Replicated Systems. In *International Conference on Computer Aided Verification*. Springer, 251–274. https://doi.org/10.1007/978-3-030-53288-8_13

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

[68] Abel Nieto, Arnaud Daby-Seesaram, Léon Gondelman, Amin Timany, and Lars Birkedal. 2023. Modular Verification of State-Based CRDTs in Separation Logic. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2023.22

[69] Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. 2022. Modular verification of op-based CRDTs in separation logic. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1788–1816. https://doi.org/10.1145/3563351

[70] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. ACM, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.62549

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

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

[73] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 472–487. https://doi.org/10.1145/3477132.3483582

[74] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653. https://doi.org/10.1145/322154.322158

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

[76] Fernando Pedone and André Schiper. 2002. Handling message semantics with generic broadcast protocols. *Distributed Computing* 15, 2 (2002), 97–107. https://doi.org/10.1007/s004460100061

[77] Nuno Preguiça, J Legatheaux Martins, Miguel Cunha, and Henrique Domingos. 2003. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st international conference on Mobile systems, applications and services*. 43–56. https://doi.org/10.1145/1066116.1189038

[78] Xukan Ran, Carter Slocum, Yi-Zhen Tsai, Kittipat Apicharttrisorn, Maria Gorlatova, and Jiasi Chen. 2020. Multi-user augmented reality with communication efficient and spatially consistent virtual objects. In *Proc. ACM CoNEXT*. https://doi.org/10.1145/3386367.3431312

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

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

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

[82] Kimberly Ruth, Tadayoshi Kohno, and Franziska Roesner. 2019. Secure Multi-User Content Sharing for Augmented Reality Applications. In *USENIX Security*.

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

[84] 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.

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

[86] Ichiro Shimoyama, Toshiaki Ninchoji, and Kenichi Uemura. 1990. The finger-tapping test: a quantitative analysis. *Archives of neurology* 47, 6 (1990), 681–684. https://doi.org/10.1001/ARCHNEUR.1990.00530060095025

[87] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

[88] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. 2022. Certified mergeable replicated data types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 332–347. https://doi.org/10.1145/3519939.3523735

[89] Ryo Suzuki, Adnan Karim, Tian Xia, Hooman Hedayati, and Nicolai Marquardt. 2022. Augmented reality and robotics: A survey and taxonomy for ar-enhanced human-robot interaction and robotic interfaces. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–33. https://doi.org/10.1145/3491102.3517719

[90] Unity. 2023. Tricks and patterns to deal with latency. https://docs-multiplayer.unity3d.com/netcode/current/learn/dealing-with-latency/index.html.

[91] Valve Developer Community. 2023. Source Multiplayer Networking. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

[92] Werner Vogels. 2008. Eventually Consistent: Building reliable distributed systems at a worldwide scale demands trade-offs?between consistency and availability. *ACM Queue* 6, 6 (2008). https://doi.org/10.1145/1466443.1466448

[93] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 980–993. https://doi.org/10.1145/3314221.3314617

[94] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. 2020. Composing and decomposing op-based CRDTs with semidirect products. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–27. https://doi.org/10.1145/3408976

[95] Michael Whittaker and Joseph M Hellerstein. 2018. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment* 12, 1 (2018), 14–27. https://doi.org/10.14778/3275536.3275538

[96] Michael Whittaker and Joseph M Hellerstein. 2020. Checking invariant confluence, in whole or in parts. *ACM SIGMOD Record* 49, 1 (2020), 7–14. https://doi.org/10.1145/3422648.3422651

[97] Haifeng Yu and Amin Vahdat. 2000. Efficient numerical error bounding for replicated network services. In *VLDB*. Citeseer, 123–133.

[98] Haifeng Yu and Amin Vahdat. 2002. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)* 20, 3 (2002), 239–282. https://doi.org/10.1145/566340.566342

[99] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-checking CRDT convergence. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1365–1388. https://doi.org/10.1145/3591276

[100] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal specification and verification of crdts. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 33–48. https://doi.org/10.1007/978-3-662-43613-4_3

Received 2024-10-14; accepted 2025-02-18