TOPOLOGY PROTOCOL: A DISTRIBUTED SYSTEM PROTOCOL FOR THE OPEN METAVERSE

Thomas (Hsea-Ching) Hsueh
Topology
gg@topology.gg
@guiltygyoza

ABSTRACT

We present Topology Protocol, a distributed system protocol designed for the Open Metaverse. The Open Metaverse is a programmable medium that enables real-time multi-user, interoperable, and censorship-resistant applications on open decentralized networks. To unlock a new phase of human productivity and collaboration, this new medium requires a distributed system protocol that is credibly neutral and scalable. It must support billions of equal participants interacting in real time at negligible costs while preserving user autonomy and privacy — fundamental needs in human interactions. To meet such requirements, we introduce a new abstraction called Conflict-free Replicated Objects (CRO) and the concept of decentralized Random Access Memory (dRAM). CROs are composable programmable objects that can be subscribed to as PubSub groups on open P2P networks. They can be mutated in real time concurrently while avoiding the costs of coordination, leveraging the properties of Conflict-free Replicated Data Types (CRDTs). Hash graphs provide CROs with causal ordering and immunity to Sybil attacks. The Topology Protocol specifies methods for CRO interaction and their corresponding behavior on P2P networks. Utility, scalability and security aspects such as snapshot, compaction, verifiability, access control and equivocation tolerance are addressed. Nodes that implement the Topology Protocol form an open ephemeral P2P network, maintaining CRO states at the edge for immediate access, and in sync across all subscriber nodes. Summarizing its capabilities, we refer to this P2P network as a dRAM, a new kind of decentralized network powering distributed programs that are lock-free concurrent. CROs on dRAM will be able to interoperate with blockchains and open standards at other layers of the computing stack, advancing the landscape of world-scale distributed computing and becoming a key building block at the foundation of the Open Metaverse.

***K*eywords** CRDTs · hash graphs · Byzantine fault tolerance · peer-to-peer networks · threshold logical clock

1 Introduction

Multi-user applications on the Internet largely rely on centralized intermediaries to mediate user interactions. While this architecture has seen tremendous success, it suffers a number of problems. Intermediaries dictate who can access what applications when and how, limiting user agency and autonomy. Interoperability among applications is rarely possible because most intermediaries operate on the business model of building and protecting their own network effects.

The Open Metaverse must operate on open decentralized networks [1]. The network architecture should be horizontally scalable. It must allow billions of participants to interact in real time at negligible costs. New multi-user software can be deployed, discovered, and accessed, completely free of censorship. As such, the Open Metaverse is much broader than video games and will maximize the freedom, creativity, and productivity of humanity on the Internet.

Blockchains laid the foundation for trust-minimized ownership and censorship-resistant multiplayer software on open decentralized networks. However, their reliance on Byzantine consensus mechanisms creates limitations in speed and costs. Operating under the strict serializability model, most modern blockchain networks require global coordination among their geo-distributed nodes to reach consistency. Yet, light only travels a few inches per clock cycle of a modern CPU. Relying on global coordination over the Internet makes the transaction throughput of blockchains heavily communication-bound. To tolerate Byzantine faults, most modern blockchains employ expensive Sybil countermeasures [2, 3], creating price floors on transaction fees to pay node operators and limiting these systems to financial use cases.

Various approaches to the blockchain scalability problem exist, yet none of them address the fundamental issue of coordination costs. One approach is to shard the network by applications [4, 5]. Another approach is to batch total-ordered blocks of transactions at rollups and submit their commitments and proofs on blockchains [6]. However, the transaction throughput of each shard or rollup remains heavily communication-bound.

In this paper, we sketch out a solution that leverages the concurrency, composability, and local-first [7] properties of Conflict-free Replicated Data Types (CRDTs) [8]. We propose Topology Protocol, a protocol dedicated to enabling applications that are real-time multiplayer, sovereign, and running on open P2P networks. We introduce Conflict-free Replicated Objects (CRO), a new abstraction for building multiplayer programs at ease and making them accessible on the P2P network. We explore how CROs can be applied to social graphs for social media applications, and multiplayer physics simulations with elastic collision constraints. We discuss various aspects that impact the flexibility, scalability and security of CROs, including how they can produce verifiable compacted snapshots for interacting with blockchains. Nodes that implement Topology Protocol form an ephemeral P2P network that keeps CRO states next to local compute on the edge, effectively forming a decentralized Random Access Memory (dRAM). Interoperating with open standards and protocols at other layers of the stack, Topology Protocol aims to grow into the de facto distributed system protocol of the Open Metaverse.

2 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) encapsulate coordination-free replication strategies and expose the application programming interface (API) of ordinary data types. Replicas of the

same CRDT can progress independently. They do not require locks for synchronization. Conflicts caused by concurrent operations are automatically resolved by rules as part of the specification of the type. As a result, all replicas are guaranteed to converge eventually without ever having to coordinate.

CRDTs are expressive. One approach is to make complex CRDTs from scratch. Although notoriously tricky to get right, correct CRDTs exist for indexed sequence types [9, 10, 11, 12], XML [13], and JSON [14]. Another approach is to make complex CRDTs through class composition. A class whose fields are all typed by CRDTs, and whose methods access its fields only through the public methods of their types, is itself a CRDT.

3 Conflict-free Replicated Objects

The primary abstraction of Topology Protocol is *Conflict-free Replicated Object* (CRO). CROs are composable programmable objects that can be updated in real time concurrently and subscribed to as publish/subscribe (PubSub) groups on open P2P networks.

Each CRO is an instance of a *blueprint*. A blueprint has two main components:

1. States: Each state is typed with either a built-in CRDT recognized by the protocol, or by another blueprint through class composition.
2. Functions: A function may read and *monotonically mutate* (*i.e.,* inflate) the states. All function signatures are specified in the blueprint's Application Binary Interface (ABI).

Additionally, a *merge* function must be defined for merging the states of an external replica with the local states. This merge function must be commutative, associative, and idempotent for convergence purposes. Developers can create blueprints in their programming language of choice. These blueprints can then be compiled into suitable bytecode formats for execution by nodes implementing the protocol.

4 Causal ordering

To further enhance the expressivity of CROs, we introduce causal ordering among their operations. This ordering allows for more complex relationships between operations. Causal order is a partial order that can be enforced in a distributed system without coordination. This differs from the total order in systems that offer strict serializability, which requires coordination. Given an operation history of a CRO, its state is derived from applying the operations in a linear order obtained from topological sort that preserves the causal order.

Approaches such as vector clocks and version vectors exist for capturing causality in distributed systems. However, these approaches are vulnerable to equivocation, making them unsafe in the presence of Byzantine actors. We need a way to provide causal ordering while tolerating Byzantine faults.

4.1 Hash graphs

We propose a solution based on hash graphs [15, 16, 17, 18]. The hash graph approach works by encoding an operation history in a directed acyclic graph, where the edges represent *causal*

dependency reporting among the operations, and vertices contain both operations and the hashes of their causal dependencies.

Notationally:

1. Let u be an update operation. Given two operations u_1 and u_2, if u_2 reported u_1 as its causal dependency, u_2 must have happened-after u_1. In the corresponding hash graph, there is an arrow pointing from u_2 to u_1, or $u_2 \rightarrow u_1$.
2. Let $h(\cdot)$ be a suitable hash function.
3. Let **F** be the hash graph's *frontier*, or the set of vertices whose operations are currently not the dependencies of any other operation. Let $\mathbf{F}^*$ be the *hashed frontier*, or $\{h(v)|v \in \mathbf{F}\}$.
4. Let **O** be the hash graph's *origin*, or the set of vertices whose operations do not causally depend on any other operation.
5. For an operation, let **D** be the set of hashed vertices that are its causal dependencies. For a newly generated operation u, its **D** equals $\mathbf{F}^*$ at the time of generation.
6. Attached to the origin of a hash graph is the *original state* of the CRO, or its state before applying any of the operations in the hash graph.

We can then define a vertex of a hash graph as containing the tuple $(u, \mathbf{D})$. The collision resistance of $h(\cdot)$ ensures that the hash graph contains no cycles. Figure 1 shows an example of a hash graph.

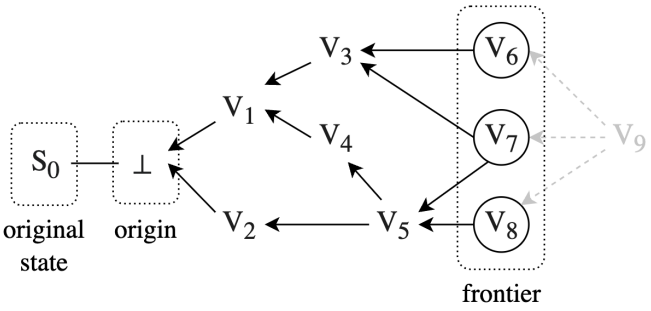

Figure 1: A hash graph of a CRO for illustration purposes. Given this hash graph, vertex V_7 should contain $(u_7, \{h(V_3), h(V_5)\})$. The vertices in the frontier $\mathbf{F} = \{V_6, V_7, V_8\}$ are circled. The origin **O** contains only one vertex, $\perp$, which carries a null operation. For the next vertex to be added to the graph, V_9, its causal dependencies should be $\mathbf{F}^* = \{h(V_6), h(V_7), h(V_8)\}$. The original state of this CRO is S_0.

Using this approach, when two nodes synchronize their operation histories of the same CRO, they effectively merge their hash graphs. Having matching $\mathbf{F}^*$ implies having equivalent hash graphs. Operations whose reported causal dependencies are unrecognized by honest nodes will not be added to their hash graphs.

As another example, consider a distributed system comprising node A and B, separated by a network delay of 40 ms. Both nodes start from the same initial state, denoted by $\perp$. Each node generates 60 operations per second ($\approx$16.6 ms between consecutive operations), a standard frame rate for games, and reconciles its hash graph with the other node as fast as the network conditions permit. Figure 2 shows the hash graphs at node A immediately before and after A generates vertex A_5.

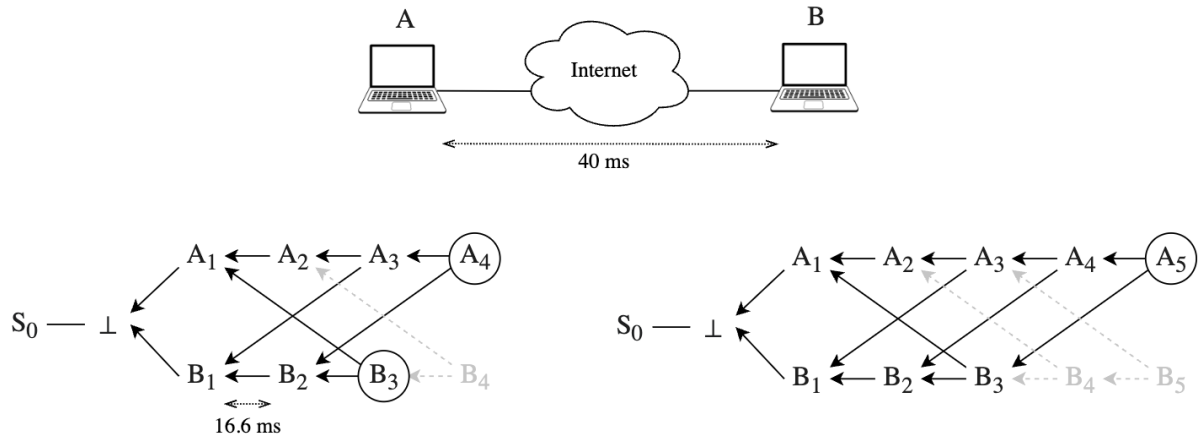

Figure 2: The hash graphs of node A in a hypothetical system immediately before and after A generates vertex A_5. Node A has not learned about the grayed out vertices, B_4 and B_5.

This approach is immune to Sybil attacks. As long as honest nodes form a connected subgraph in the P2P network, the system is able to function correctly. This allows CROs to tolerate arbitrarily many Sybil actors, hence the immunity. In contrast, systems with global Byzantine consensus functions correctly only if less than one third of the nodes are faulty. No expensive countermeasures such as Proof of Work or Proof of Stake are involved, making the costs of transacting with CROs practically zero.

4.2 Concurrency semantics

Equipped with causal ordering, we now have a notion of concurrency. Two operations can be comparable (*i.e.* one happened after the other) or incomparable (*i.e.* they are concurrent to each other). For operations that happened concurrently but do not commute, the order at which they are serialized and executed impacts the CRO state. The *concurrency semantics* of a CRO define its behavior in the presence of concurrent operations that are not commutative.

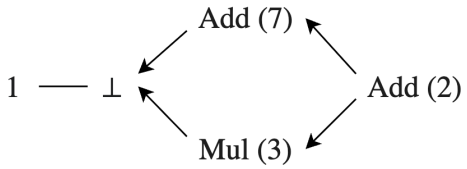

Figure 3: A hash graph for the register CRO that accepts addition and multiplication.

Consider a register CRO that holds a single integer and accepts two types of operations:

1. $Add(x)$: adds x to the current value
2. $Mul(x)$: multiplies the current value by x

However, integer addition and multiplication do not commute. Figure 3 shows a hash graph for this register. We can see two possible execution orders:

1. $(1 + 7) * 3 + 2 = 26$
2. $(1 * 3) + 7 + 2 = 12$

To resolve this ambiguity, we can introduce the following concurrency semantics to the CRO: addition goes first in case of concurrency. With this rule, every honest replica of this hash graph would arrive at 26 as its final state.

5 Signaling

Having established how individual CROs function, we now turn to how these objects interact with each other. CROs interact by *signaling*, a CRO-level term to differentiate from messaging at the network layer. A signal originates from an operation at the sender CRO. It then materializes into an operation at the receiver CRO. For causality to work, a signal needs to specify its causal dependencies in both the sender CRO and receiver CRO's hash graphs. This means the node that intends to send a cross-object signal needs to have the hash graphs of both the sender and receiver CROs locally. This requirement creates challenges in scalability, as nodes must maintain and synchronize multiple hash graphs simultaneously. The safety and scalability of cross-object signaling remain an active area of research.

Figure 4 illustrates a cross-object signal diagrammatically, where the signal "glues" together its originating operation on the sender CRO side and the operation it materializes into on the receiver CRO side.

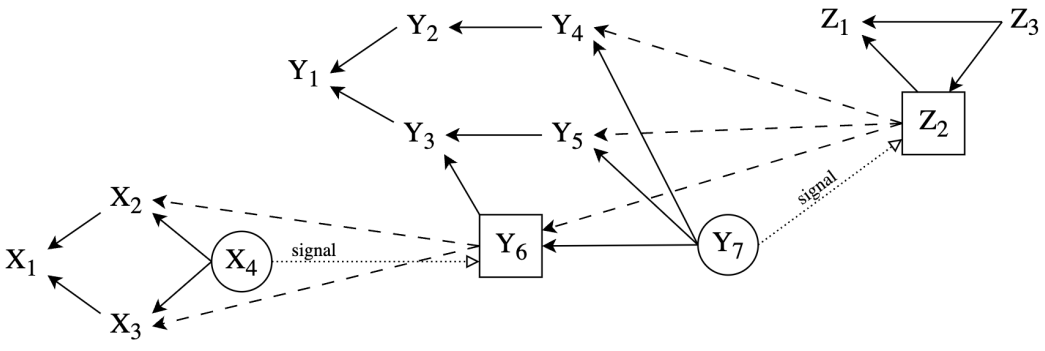

Figure 4: Hash graphs of CRO X, Y, Z interacting by passing signals (original states are omitted). Cross-object signals are denoted as dotted arrows. Cross-object causal dependencies are denoted as dashed pointers. The vertices where signals are originated are enclosed in circles, whereas those on the receiving sides are enclosed in squares. In this example, the generation of vertex X_4 also produces a signal to object Y, materializing into vertex Y_6. Signaling also occurred between vertices Y_7 and Z_2. The causal dependencies of Y_6 is X_2, X_3, and Y_3. The causal dependencies of Z_2 is Y_4, Y_5, Y_6, and Z_1.

6 Invariants

Besides composability, preservation of invariants of all kinds is crucial for CROs to be richly expressive. An invariant in a distributed system is a predicate over the system state or state transitions that is always true, observed at any of its replicas. Consider two types of invariants:

1. *Numeric invariants*. For example, a numeric invariant required for cryptocurrency applications is that an account balance is nonnegative at all times. This invariant could break down when we allow concurrent updates to the same account — a shared wallet with $1 left, spent concurrently by Alice and Bob.
2. *Integrity invariants*. For example, referential integrity requires that if there is a pointer from object A to B, B must be valid. This invariant is compromised when a new reference to B is added while B is invalidated concurrently.

Common approaches to preserving such invariants include:

1. *Reservation*. Reservation assigns each replica the right to perform certain operations in advance [19]. For example, a wallet of an initial balance $2N may give Alice and Bob each the right to spend exactly $N.
2. *Compensation*. Compensation repairs the violation of invariants by performing additional operations [20]. Using the same wallet example, when a replica detects a negative balance, it may generate a compensation operation that invalidates one or some of the latest spend operations following a deterministic policy, restoring the account balance back to nonnegative. All honest replicas follow the same policy, guaranteeing convergence. In the case of referential integrity, when a replica detects the invalidated object B was concurrently referred to by another operation, it may generate a compensation operation that restores the validity of object B but sets its content to null.

7 Examples

In this section, we provide two examples that illustrate the expressivity of CRO.

7.1 Example 1

The first example is a social graph that demonstrates the composability of CROs. Algorithm 1 shows the pseudocode of a blueprint named GMapGSet, which is a grow-only map (GMap) where values are grow-only sets (GSets). When adding a key-value pair to it, the pair is inserted into the map if the key is absent, otherwise the value is added to the GSet corresponding to the key in the map.

We can build a basic social graph on top of this CRO. Algorithm 2 shows the pseudocode of a blueprint named SocialGraph, which uses a GSet to store user public keys, a two-phase set (2PSet) to store posts, and instantiates the GMapGSet blueprint to store the timestamps when users are added. Note that the timestamps in the signals $AddUser$ and $AddPost$ are reported by nodes voluntarily, which can be faulty. In the next example, we drop these timestamps and leverage hash graphs for causal ordering.

Algorithm 1 GMapGSet

state:
Map : GMap<any ⇒ GSet>

interface:
signal $Add\,(k : \text{any}, v : \text{any})$
function $Get\,(k : \text{any}) \rightarrow\ v$: Set<any>

on signal Add
$k \notin map \Rightarrow map.insert(k, v)$
$else \Rightarrow map[k].add(v)$

function Get
return $map[k]$

7.2 Example 2

The second example is a multiplayer physics simulation that demonstrates the compensation technique and the use of hash graph. For simplicity, we construct a 2D space where players move around in the form of circle objects that collide elastically with one another. These circle objects have random radii and masses, whose values are determined when players join the space. The physics state of each object contains a position vector and a velocity vector. The goal is to design a blueprint that describes this construction.

We start by specifying two types of operations, *Drive* and *Tick*. Drive applies a change to the velocity vector of an object. Tick advances the time of an object, which causes its position to change by velocity times a constant time difference. Objects are driven and ticked only by the players that control them.

Traversing a hash graph containing such operations, we can obtain for each player a sequence of operations they generated. To compute the position of a particular player, we traverse their sequence of operations, updating the velocity when we encounter a Drive operation, and updating the position by adding a displacement to it when we encounter a Tick operation. The displacement is equal to the product of the velocity and the constant time difference.

However, the above construction does not consider the elastic collision constraints. Geometrically, a collision between two objects occurs when their shapes overlap. Equation 1 expresses the no-overlap constraint for two circles i and j, where (x, y) refers to the center of a circle, and r refers to its radius. Clearly, this is a numeric invariant that can be violated by concurrent positional changes of the circles. Equation 2 expresses the elastic collision constraint between two circles i and j, where their relative velocites along the normal direction before and after the collision are v_{rel}^{-} and v_{rel}^{+} respectively, and e is a scalar representing the restitution coefficient.

$$(x_i - x_j)^2 + (y_i - y_j)^2 \ \geq\ (r_i + r_j)^2 \tag{1}$$

$$v_{rel}^{+} = -e \cdot v_{rel}^{-} \tag{2}$$

Algorithm 2 SocialGraph

state:

$Users$: GSet<User> ▷ Grow-only set; built-in
$Posts$: 2PSet<Post> ▷ Two-phase set; built-in
$UserTimestamps$: GMapGSet ▷ The blueprint from Algorithm 1

interface:

signal $AddUser\,(userPublicKey$: User, $timestamp$: Timestamp)
signal $AddPost\,(post$: Post)
signal $RemovePost\,(post$: Post)
function $GetUsers\,()$ → Set<User>
function $GetPosts\,()$ → Set<Post>

on signal $AddUser$
$Users.Add(userPublicKey)$
$UserTimestamps.Add(userPublicKey,\ timestamp)$

on signal $AddPost$
$Posts.Add(post)$

on signal $RemovePost$
$Posts.Remove(post)$

function $GetUsers$
return $Users.Set$

function $GetPosts$
return {
$p, \forall p \in Posts.AddSet \ \wedge \ p \notin Posts.RemoveSet \ \wedge \ p.user \in Users.Set$
$\wedge\, p.timestamp > UserTimestamps.Get(p.user).max()$
}
▷ the post needs to (1) survive in the add set,
▷ (2) be added by a recognized user after the user was added

function $Merge\,(replica)$
$Users \leftarrow Users \cup replica.Users$ ▷ union operator is defined for 2PSet
$Posts \leftarrow Posts \cup replica.Posts$

Algorithm 3 Multiplayer2DPhysics

state:

$history$: HashGraph

interface:

signal $Drive\,(id$: Player, dv : Vec2) ▷ Vec2 is a 2d vector of numbers

signal $Tick\,(id$: Player)

function $GetPlayerPhysicsStates\,()$ → Map<Player⇒Vec2>, Map<Player⇒Vec2>

▷ One map for positions, the other map for velocities

on signal $Drive$

$history.Add(Drive(id, dv))$

on signal $Tick$

$history.Add(Tick(id))$

$\_EnforceConstraints()$

function $\_EnforceConstraints$

$P, V \leftarrow GetPlayerPhysicsStates()$

if $\neg$*_hasOverlap*(P) **return** ▷ *_hasOverlap* checks for overlapping circles

$V_{comp} \leftarrow \_ComputeCompensation(P, V)$

$\forall (id, v) \in V_{comp}\,\{$

$history.Add(Compensate(id, v))$

$\}$

function $GetPlayerPhysicsStates$

$operations \leftarrow history.serialize(Compensate > Drive, Tick)$

▷ serialize by topological sort; $Compensate$ wins in case of concurrency

$P \leftarrow InitialPlayerPositions$

$V \leftarrow InitialPlayerVelocities$

$\forall op \in operations\,\{$

$type(op) = Drive \Rightarrow V[op.id] \leftarrow V[op.id] + op.dv$

$type(op) = Compensate \Rightarrow V[op.id] \leftarrow op.v$

$type(op) = Tick \Rightarrow P[op.id] \leftarrow P[op.id] + V[op.id] * DT$

▷ DT is the constant time difference

$\}$

return P, V

function $Merge\,(replica)$

$history \leftarrow history \cup replica.history$ ▷ union operator is defined for HashGraph

$\_EnforceConstraints()$

We adopt the compensation technique to enforce both the no-overlap constraint and the perfect elastic collision constraint. A new type of operation is introduced, *Compensate*, which directly sets the velocities of the colliding circles to new values in order to repair constraint violation. To ensure convergence, we also introduce the semantics that *Compensate* wins in case it is concurrent to *Drive* or *Tick* that attempts to mutate the velocities of the same circles. Many approaches exist for computing these compensating velocity values, such as the sequential impulse approach [21, 22] and the Linear Complementarity Problem (LCP) approach [23, 24]. The details of these approaches are outside of the scope of this paper. This construction has a number of issues, including tunneling (*i.e.,* when two objects pass through each other without overlapping at any tick) and players ticking at different rates. How to modify the construction to handle these issues is beyond the scope of this paper.

Algorithm 3 shows the pseudocode of a blueprint named Multiplayer2DPhysics, illustrating the construction above. By having $\_ComputeCompensation$ a fully deterministic procedure, we can ensure that all honest replicas will generate the exact same compensations, guaranteeing convergence.

8 Interaction methods

Topology Protocol defines a set of methods, or simply *verbs*, for nodes to interact with a given CRO. At the core of the protocol is the PubSub model for nodes in P2P networks to subscribe to CROs and publish updates on them asynchronously. CROs are identified as PubSub groups or topics. Nodes only subscribe to CROs they are interested in. This keeps CROs loosely coupled and helps reduce memory and bandwidth requirement for operating nodes. In contrast, every network participant of a blockchain is by default "subscribed" to all smart contracts. Figure 5 illustrates this interaction pattern.

The set of core interaction methods and their semantics is as follows:

1. **CREATE**: to create a new CRO. Under the hood, a PubSub group (topic) is created for the new CRO.
2. **UPDATE**: to perform an update operation on a given CRO. The update is published to the corresponding PubSub group.
3. **SUBSCRIBE**: to subscribe to all updates performed on a given CRO. The node adds the corresponding PubSub group to its groups of interest.
4. **UNSUBSCRIBE**: to unsubscribe and stop receiving updates on a given CRO. The node removes the corresponding PubSub group from its groups of interest.
5. **SYNC**: to reconcile the differences in operation histories (hash graphs) of a given CRO between the local one and a remote one. The SYNC method has two main purposes: for a new node to bootstrap into an existing CRO, and as an out-of-band synchronization method besides P2P gossiping.

SYNC can take a long time to complete when a node is bootstrapping into a CRO with a large history. This can be viewed as a problem of set reconciliation. One common approach is to exchange bloom filters to reduce the message complexity of the sync process. A more promising approach has constant overhead even when reconciling sets with very large differences [25].

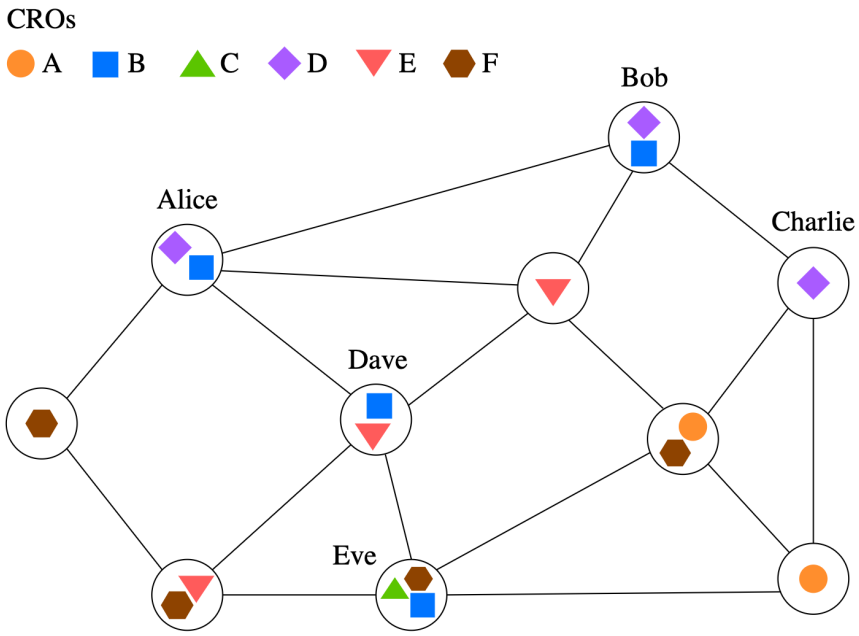

Figure 5: Alice, Bob, and Charlie are subscribed to CRO D. Alice, Bob, Dave, and Eve are subscribed to CRO B. All nodes are connected peer-to-peer.

9 Operation life cycle

The life cycle of a CRO operation differs significantly from a blockchain transaction. When a node performs a write (update) by generating an operation on a CRO, the operation is added to the node's local copy of the CRO's hash graph immediately. A node reads a CRO state by serializing the node's local copy of the CRO's hash graph. If a node performs a write and a subsequent read immediately, the read is guaranteed to observe the write. In other words, CRO provides highly available operations and guarantees low latency [26]. In contrast, blockchains are not highly available. A blockchain transaction becomes available only after it is included in a block that is proposed, verified and acknowledged by a threshold amount of network participants.

10 Snapshot

A CRO snapshot is a single hash graph that represents an agreement among the CRO's replicas. It serves multiple purposes:

1. It acts as a "state save" that may be worth persisted (*e.g.*, on IPFS [27]).
2. It can be used to generate irreversible transactions on blockchains.

The computation of snapshots faces several problems. We must avoid involving "all nodes" in the algorithm: unanimity can be very problematic due to network conditions, dynamic membership, and Byzantine behavior. The algorithm needs to run concurrently to the underlying CRO activities without hindering or suspending them. The algorithm must complete with reasonable bandwidth consumption, thus the nodes involved must not send entire replicas over the wire.

Our current solution builds on threshold logical clock (TLC) [28] as a decentralized pacemaker, on top of which Byzantine consensus can be reached. Each CRO has its own TLC operated by its subscriber nodes. When a node is ready to advance the TLC tick, it proposes the hashed frontier $\mathbf{F}^*$ of its own hash graph by broadcasting it to other subscribers. Instead of unanimity, the tick advances by threshold amounts of subscriber nodes signing and acknowledging messages. Three consecutive ticks can be used to form a consensus round, yielding a snapshot. The process runs continuously, allowing snapshots to be taken periodically.

CRO snapshots remain an active area of research. The algorithms and their configurability supported by the protocol remain to be determined through the protocol development process.

11 Compaction

Grow-only hash graphs are problematic. Unbounded memory is needed, and large graphs take longer to bootstrap. Compaction serializes the state of a CRDT by applying the operations carried by a portion of the hash graph before pruning that portion away. Compaction discards causal information and thus is perfectly safe only when performed over vertices that will never be the causal dependencies of any future operations.

However, perfectly safe compaction can be impractical. To be certain that a vertex V is perfectly safe to drop, a node needs to receive a vertex from every other node that causally depends on V [29]. This involves the notion of "all nodes", our usual suspect. Our current solution is to use the TLC consensus rounds to drive compaction. Each round yields a hash graph, whose frontier **F** comprises vertices that are considered causally stable. All preceding vertices can be compacted. The CRO's state is forwarded by applying the operations of those compacted vertices in a linear order obtained from topological sort. Figure 6 illustrates this process.

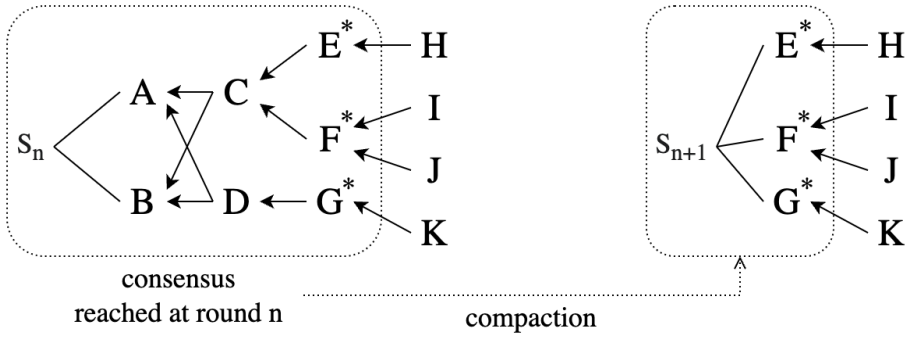

Figure 6: Compaction is performed on the subgraph containing vertices A, B, C, D that precedes the frontier **F** of the hash graph decided by consensus. The vertices in the frontier **F** of the hash graph before compaction are marked with asterisks $*$, which form the origin **O** of the hash graph after compaction. S_{n+1} is computed by applying the operations in the subgraph on S_n in a linear order obtained from topological sort.

Unsafe compaction is named as such because it may drop causal information in the hash graph that is needed to recognize the causality of operations that have yet to arrive from the network. Thus, operations with unrecognizable causality could come from both honest and Byzantine

nodes. Existing approaches from Tusk [30] and Bullshark [31] choose to re-inject transactions in a later consensus round. These approaches aim to preserve some notion of fairness, ensuring that transactions generated by correct nodes are eventually delivered to all other correct nodes. However, re-injected transactions lose their original causal dependencies, requiring rewriting on top of the hash graph frontier from the previous consensus round. A consistent, deterministic policy for rewriting dependencies is needed. Compaction remains an active area of research.

Revisiting the snapshot problem, we can see why producing grow-only snapshots is undesirable. Using compaction, we can shrink the size of a snapshot. This is particularly valuable when snapshots are to be stored on blockchains, where storage is expensive.

By leveraging recursive zero-knowledge proofs [32], it is possible to produce compacted snapshots that are verifiable, which are useful in scenarios where trust needs to be minimized. A compacted snapshot is verifiable when anyone can verify that (1) consensus was reached correctly (2) compaction was performed correctly. For the consensus part, we can require the proposals in each round to contain the hash of the agreed upon proposal and the aggregated signatures from the previous round. This effectively forms a chain. Verifying the aggregated signatures at the tip of the chain verifies the entire chain. For the compaction part, we can require the blueprint to be executable by a provable VM such as Cairo [33], RISC0 [34], or Valida [35]. The state forwarding step of compaction can be proven recursively, yielding a recursive proof that sits alongside the snapshot.

12 Access control

Access control is important for multiplayer applications. Social and communal spaces are much more useful when fine-grained permissions can be assigned to participants. Common access control models include level-based (lattice-based) [36] and role-based ones [37]. It gets tricky when a CRO's access control logic is specified within itself or another CRO, due to eventual consistency. What happens when an operation that revokes the access role of a participant is concurrent to an operation generated by the same participant that depends on having the very role assigned?

The core issue here is conflict resolution strategies that involve permission changes. A promising solution is proposed in [38], where both permission roles or levels and the causal order among operations are taken into account to topologically sort a hash graph. The solution works under the assumption that participants with higher permission roles or levels are not attackers. Another promising direction is to leverage the TLC to establish a linear order between permission-changing operations and permission-dependent operations. A CRO may require that permission-changing operations remain ineffective until the next TLC tick.

13 Security

Security is a critical aspect of CROs. We consider three threat models for CROs and explore potential solutions.

13.1 Hash graph pollution

Hash graphs can be polluted by vertices with garbage payload such as invalid operations. When receiving a new vertex, a node can validate its operation before adding it to the local hash graph.

The nodes that published bad operations can be banned. If an operation can be validated atomically, we can be certain that all honest nodes will reach the same conclusion. It is trickier when the validity of the operation depends on the operations that happened before it. In this case, we can define operation validity as a function of the operation itself and all operations that can be transitively reached through the causal dependencies in the hash graph [16].

13.2 Equivocation

Equivocation occurs when a node, assumed to be a single-threaded process, generates concurrent operations. An honest node can detect that another node has generated concurrent operations. The honest node may choose to remove the corresponding bad vertices from its hash graph, ban the Byzantine node, and expect all other honest nodes to do the same. However, downstream vertices that causally depend on these bad vertices may have been generated. It is not obvious how to graft these downstream vertices back to the hash graph. Equivocation tolerance remains an active area of research.

13.3 Backdated dependencies

We consider this as the trickiest threat model: when nodes report dependencies that are older than the vertices in the frontier. The question is how backdated dependencies can be reliably detected by all honest nodes. A vertex carrying correct dependencies but arriving late due to network delays is indistinguishable from a vertex with backdated dependencies. One promising direction is to leverage the TLC as it introduces a notion of linear time. Denote the frontier of the hash graph from n consensus rounds ago as $\mathbf{F}_{-n}$. A CRO can stipulate that a vertex should not report causal dependencies that causally precede any of the vertices in $\mathbf{F}_{-n}$ with n carefully chosen, otherwise the vertex will be handled according to some policy that is applied consistently across all honest nodes. As such, the TLC serves as a security clock for the CRO, whose $\mathbf{F}_{-n}$ serves as the backstop for backdated dependencies.

14 Decentralized random access memory

A compelling perspective on Topology Protocol is its ability to power a decentralized network that functions as a "decentralized Random Access Memory" (dRAM). The following properties of the protocol support this perspective:

1. **Random access**. Users can synchronize on any CROs of interest without having to synchronize every single CRO in the system, contrary to smart contracts on blockchains. This access pattern reduces the resource requirement of running a protocol node, making it more accessible for users to participate in the network.
2. **Closeness to compute**. We envision that most users of CRO-powered applications will operate nodes on their devices, holding CRO replicas locally. In other words, the protocol champions the principles of local-first software [7]. This approach reduces latency, enhances privacy, and improves both scalability and resilience by "load-balancing" across user devices. In contrast, most blockchain users access smart contracts via centralized RPC nodes, not unlike the access pattern of most cloud apps. This property is analogous to the role random access memory serves in a memory hierarchy.

3. **Ephemerality**. Persistence of CROs is out of scope of the protocol. The protocol expects the nodes to be ephemeral. Their states are volatile from the system's perspective, corresponding to the volatility of RAM. The data movement between Topology nodes and persistence providers is analogous to the data movement between RAM and disk storage which is non-volatile.

Decentralized RAM has the potential to enable a new class of decentralized applications that can operate more dynamically and responsively than blockchain-based applications.

15 Conclusion

In this paper, we introduced CRO and Topology Protocol for P2P real-time multiplayer applications. CRDT is a natural choice for such use cases, and the use of hash graphs allows capturing causality among operations while tolerating Byzantine faults. CRO provides the programmability of common object systems and encapsulates coordination-free replication strategies. Topology Protocol specifies a set of interaction methods for CROs to be created, updated, subscribed to, unsubscribed from, and synchronized. Their semantics correspond to behavior on the underlying P2P networks under the publish/subscribe model. Snapshot, compaction, access control, and threat models of CROs were addressed. Nodes that implement Topology Protocol keep replicas of CROs locally, facilitate persistence without being primarily responsible for it, and disseminate CRO updates only to nodes that are interested in them. As such, a P2P network comprised of such nodes effectively works as a decentralized RAM.

Acknowledgments

The development of this work has been greatly enriched by the insights and support of many individuals. We would like to express our gratitude to:

- Eli Ben-Sasson and Uri Kolodny, for supporting us in our endeavors to run scalable physics simulations in a decentralized manner;
- Joon Yun, for pointing us toward the magic of CRDTs in the first place;
- Martin Kleppmann, for the discussions on the theory behind CRDTs and the mechanics and threat models of hash graphs;
- Zaki Manian, for discussions on the solutions to P2P connectivity issues;
- Sina Habibian, for encouraging us to explore alternative distributed data structures on P2P networks that provide different affordances from what blockchains offer;
- Polynya, for encouraging us to explore new software paradigms that promote the creative agency of a broad audience, and to seek alternatives to the plutocracy and oligopoly that underlie the correct functioning of all major blockchains;
- Jihoon Song and Kunho Kim, for the original discussions that motivated this paper;
- Jay Oak, for the pivotal input in shaping the scope and capabilities of the protocol and the abundant feedback for improving this paper;
- Akash S M, for suggesting that we explain why concurrency semantics are important;

- Matthew Weidner, for pointing out the challenges in exercising access control under eventual consistency, and a solution designed for the Matrix Protocol;
- Peter van Hardenberg, for pointing out the importance of invariant preservation and suggesting that we add a representative example for discussing how constraints may be enforced in CROs;
- Pierre Semanne and Goblin Oats, for advocating against a global finality system shared by all CROs, an idea explored in the early drafts;
- haadcode, for reinforcing our concept of a decentralized RAM;
- Oskar Thorén and Anderson Chen, for carefully reviewing the early drafts and offering ideas for improvement;
- Alok Vasudev and Adam Goldberg, for the patience and encouragement for us to focus on finding the truth.

References

[1] SIGGRAPH Conferences. SIGGRAPH Spotlight: Episode 30 – Tim Sweeney and the Metaverse. ACM SIGGRAPH, 2019.

[2] Jae Kwon. Tendermint: Consensus without Mining. Cornell University, 2014.

[3] Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[4] Jae Kwon and Ethan Buchman. Cosmos: A Network of Distributed Ledgers. Tendermint, 2016.

[5] Maxim Orlovsky, Peter Todd, Giacomo Zucco, Federico Tenga, and Olga Ukolova. RGB Blackpaper. 2023.

[6] Vitalik Buterin. An Incomplete Guide to Rollups. 2021.

[7] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. Ink Switch, 2019.

[8] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. 2011.

[9] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. *13th ACM Symposium on Document Engineering (DocEng)*, pages 37–46, 2013.

[10] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing, Volume 71, Issue 3*, pages 354–368, 2011.

[11] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *Proceedings of the 2016 ACM International Conference on Supporting Group Work*, pages 39–49, 2016.

[12] Matthew Weidner and Martin Kleppmann. The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing. *arXiv preprint arXiv:2305.00583*, 2023.

[13] Stéphane Martin, Pascal Urso, and Stéphane Weiss. Scalable XML Collaborative Editing with Undo short paper. *arXiv preprint arXiv:1010.3615*, 2010.

[14] Martin Kleppmann and Alastair R. Beresford. A Conflict-Free Replicated JSON Datatype. *arXiv preprint arXiv:1608.03960*, 2016.

[15] Martin Kleppmann and Heidi Howard. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. *arXiv preprint arXiv:2012.00472*, 2020.

[16] Martin Kleppmann. Making CRDTs Byzantine fault tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 8–15, 2022.

[17] Héctor Sanjuán, Samuli Pöyhtäri, Pedro Teixeira, and Ioannis Psaras. Merkle-CRDTs: Merkle-DAGs meet CRDTs. *arXiv preprint arXiv:2004.00107*, 2020.

[18] Paulo Sérgio Almeida and Ehud Shapiro. The Blocklace: A Universal, Byzantine Fault-Tolerant, Conflict-free Replicated Data Type. *arXiv preprint arXiv:2402.08068*, 2024.

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

[20] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. IPA: Invariant-preserving Applications for Weakly-consistent Replicated Databases. *Proc. VLDB Endow.*, 12(4):404–418, dec 2018.

[21] Thomas Jakobsen. Advanced character physics. In *Game developers conference*, volume 3, pages 383–401. IO Interactive, Copenhagen Denmark, 2001.

[22] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid bodies with stacking. *ACM transactions on graphics (TOG)*, 22(3):871–878, 2003.

[23] David Baraff. Non-penetrating rigid body simulation. *State of the art reports*, 1993.

[24] Erin Catto. Iterative dynamics with temporal coherence. In *Game developer conference*, volume 2, 2005.

[25] Lei Yang, Yossi Gilad, and Mohammad Alizadeh. Practical Rateless Set Reconciliation. *arXiv preprint arXiv:2402.02668*, 2024.

[26] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, nov 2013.

[27] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. *arXiv preprint arXiv:1407.3561*, 2014.

[28] Bryan Ford. Threshold Logical Clocks for Asynchronous Distributed Coordination and Consensus. *arXiv preprint arXiv:1907.07010*, 2019.

[29] Carlos Baquero, Paulo Sergio Almeida, and Ali Shoker. Pure Operation-Based Replicated Data Types. *arXiv preprint arXiv:1710.04469*, 2017.

[30] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. *arXiv preprint arXiv:2105.11827*, 2021.

[31] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT Protocols Made Practical. *arXiv preprint arXiv:2201.05677*, 2022.

[32] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 111–120, New York, NY, USA, 2013. Association for Computing Machinery.

[33] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021.

[34] Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. RISC Zero zkVM: Scalable, Transparent Arguments of RISC-V Integrity, 2023.

[35] Valida. Valida: A STARK-based VM focused on code reuse, performance, and modularity. Github repository, 2023.

[36] Ravi S. Sandhu. Lattice-Based Access Control Models. *Computer, 26(11)*, pages 9–19, 1993.

[37] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *Computer, 29(2)*, pages 38–47, 1996.

[38] Florian Jacob, Luca Becker, Jan Grashöfer, and Hannes Hartenstein. Matrix decomposition: Analysis of an access control approach on transaction-based dags without finality. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, page 81–92, 2020.