DISTRIBUTED REAL-TIME PROGRAMS: AN OPEN INTERNET PROTOCOL FOR THE WORLD COMPUTER

Topology Inc.
www.topology.gg

ABSTRACT

We present Distributed Real-time Programs (DRP), an open peer-to-peer Internet protocol to introduce RAM-like capabilities to the World Computer. To effectively manage the vast number of real-time interactions among billions of users, the World Computer needs a new Internet protocol capable of handling concurrent operations with minimal latency and overhead, while preserving user autonomy and privacy. Programs built with DRP are concurrently writeable in real time, completely avoiding consensus overhead by leveraging the properties of Conflict-free Replicated Data Types (CRDTs). They are identified as PubSub groups on an open P2P network comprised of DRP nodes. They use hash graphs as causal logs, granting them immunity to Sybil attacks with no involvement of financial incentives. DRP specifies a set of methods for clients to indicate the actions they would like to perform on a given DRP program. Utility, scalability, and security aspects, such as snapshot, compaction, verifiability, access control, and equivocation tolerance, are addressed. We refer to the DRP network as decentralized RAM, a new kind of decentralized network powering distributed programs that are lock-free concurrent. Programs on the DRP network can 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 world computer.

Keywords 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 idea of the world computer is to create a single programmable medium where code and data live and operate correctly forever without centralized intermediaries, enabling trustless and censorship-resistant interaction at global scale. The world computer should possess the following properties:

1. Operate on decentralized networks.
2. Be horizontally scalable.
3. Allow billions of participants in interact in real time at negligible costs.
4. Allow multi-user software to be deployed, discovered, and accessed, completely free of censorship.

Blockchains laid the foundation for trust-minimized ownership and censorship-resistant multi-player software on open decentralized networks. However, their reliance on Byzantine consensus mechanisms creates limitations in both transaction speed and costs of transaction fees. Operating under the strict serializability model, most modern blockchain networks require global coordination among their geo-distributed nodes to reach consistency. Relying on global coordination over the Internet makes the transaction speed of blockchains heavily communication-bound. To tolerate Byzantine faults, most modern blockchains employ Sybil countermeasures that involve the staking of cryptocurrencies by validator nodes, and the incentivization of such act [1, 2]. Transaction fee distribution is the common source of income for validators that is more stable and predicable than inflation. Yet, the necessity to compensate validators through transaction fees imposes a minimum threshold on transaction costs. If transaction fees fall below a certain level, validators may find their incentives to participate in the network significantly diminished, potentially compromising the network's security.

Various approaches to the blockchain scalability problem exist, yet none of them address the fundamental issue of coordination costs. One approach is to endow each application with its own network [3, 4]. However, the transaction throughput of each network remains heavily communication-bound. Another approach is to batch total-ordered blocks of transactions at rollups and submit their commitments and proofs on blockchains [5]. Yet, most rollups employ centralized servers for sequencing transactions, an architecture that introduces additional trust assumption on the liveness of their systems.

In this paper, we propose **Distributed Real-time Programs** (**DRP**), an Internet protocol dedicated to applications that are real-time multiplayer, sovereign, and running on open P2P networks. The design of the protocol leverages the concurrency, composability, and local-first [6] properties of Conflict-free Replicated Data Types (CRDTs) [7]. The P2P network comprising of nodes that implement the protocol functions as a decentralized "**random access memory**" — a distributed state that is fast, ephemeral, sharded, and distributed as copies residing directly on end-user devices, located alongside their computation capabilities. Interoperating with open standards and protocols at other layers of the stack, DRP aims to become a fundamental component of the world computer architecture.

The remainder of this paper is organized as follows. Section 2 introduces the concept of DRP Objects. Section 3 discusses the use of hash graphs to ensure causal consistency and Byzantine fault tolerance. Section 4 explores concurrency semantics and the resolution of conflicts in DRP Objects. Section 5 presents the signaling mechanism, enabling DRP Objects to interact reliably. Section 6 examines invariant preservation techniques. Section 7 provides a basic example to illustrate DRP's expressivity. Section 8 presents the interaction methods within the protocol. Sections 9 through 12 cover the operation lifecycle, access control, finality, and mechanisms for snapshotting and compaction. Section 13 addresses security considerations, followed by Section 14, which introduces the notion of decentralized RAM as a broader conceptual framework. We conclude the paper in Section 15.

2 DRP Objects

DRP Objects are composable programmable objects that can be (1) updated in real time concurrently, and (2) subscribed to by interested parties individually on the open P2P network comprising of DRP nodes.

Each DRP Object has an instance of a *DRP*. The DRP has two main components:

1. *States*: States is typed with either a built-in type of the programming language in use, or another DRP through class composition.
2. *Functions*: A function may read and *monotonically mutate* (*i.e.,* inflate) the states.

Developers can create blueprints in their programming language of choice.

3 Causal ordering

To further enhance the expressivity of DRP Objects, we introduce causal ordering among their 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 DRP Object, 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.

3.1 Hash graphs

We propose a solution based on hash graphs [8, 9, 10, 11]. 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_1 to u_2, or $u_1 \rightarrow u_2$.
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 **F*** be the *hashed frontier*, or $\{h(v)|v \in \mathbf{F}\}$.
4. Let Φ be the hash graph's *root vertex*, or the vertex who does not causally depend on any other vertex.
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 **F*** at the time of generation.
6. Attached to the root vertices of a hash graph is the *initial state* of the DRP Object, or its state before applying any of the operations in the hash graph.

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

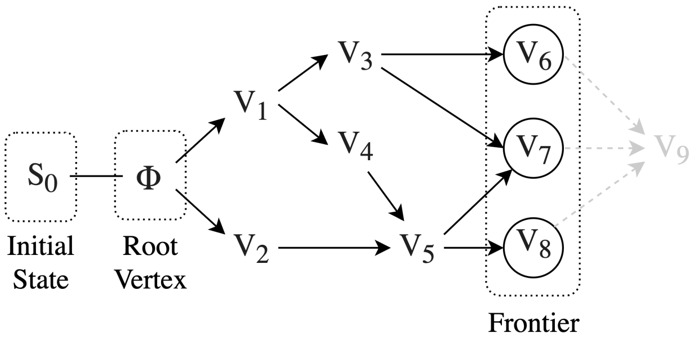

Figure 1: A hash graph of a DRP Object for illustration purposes. Given this hash graph, vertex $V_7 = (u_7, \{h(V_3), h(V_5)\})$. The vertices in the frontier $\mathbf{F} = \{V_6, V_7, V_8\}$ are circled. There is one root vertex, Φ, which carries a null operation. For the next vertex to be added to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

Using this approach, when two nodes synchronize their operation histories of the same DRP Object, they effectively merge their hash graphs. Having matching **F*** implies having equivalent hash graphs. When an honest node receives a vertex, if not all of its reported causal dependencies are recognized, the node start a sync process to resolve the unknown dependencies.

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 root vertex Φ. Each node generates 60 operations per second ($\approx$16.6 ms between consecutive operations). Operations propagate through the network to the other node and get merged into their hash graph. No consensus overhead is incurred in the process. Figure 2 shows the hash graphs at node A immediately before and after A generates vertex A_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 DRP Objects to tolerate arbitrarily many Sybil actors, hence the immunity. In contrast, systems with global Byzantine consensus functions correctly only if at most one third of the nodes are faulty.

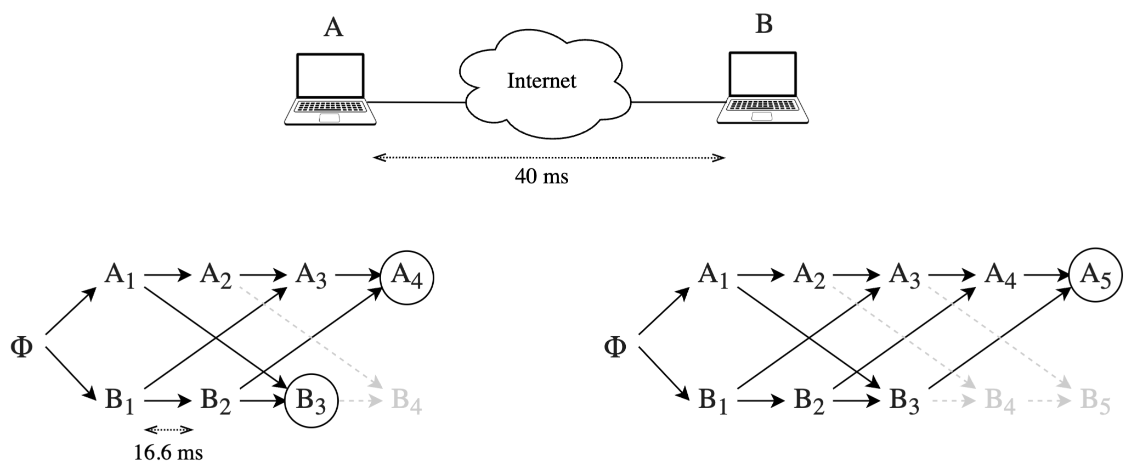

Figure 2: The hash graphs of node A immediately before and after A generates vertex A_5. Node A has not learned about the grayed out vertices, B_4 and B_5.

3.2 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 state of the DRP Object. The *concurrency semantics* of a DRP Object define its behavior in the presence of concurrent operations that do not commute.

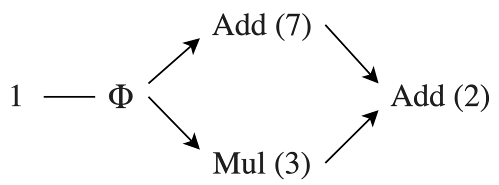

Figure 3: A hash graph for DRP Object that holds a single integer and accepts addition and multiplication.

Consider a DRP Object 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, a valid semantics can be introduced: addition goes first. With this rule, every honest replica of this hash graph would arrive at 26 as its final state.

Note that there are many possible semantics that are equally valid. The rules of thumb for choosing the right semantics are:

1. They should be *deterministic*.
2. They should *preserve user intent* as much as possible. In other words, the semantics should lead to program behaviors that make the most sense for users who interact with the program concurrently.

4 Signaling

Having established how individual DRP Objects function, we now turn to how these objects interact with each other. DRP Objects interact by *signaling*, a DRP Object-level term to differentiate from messaging at the network layer.

A signal:

1. originates from an operation, called **source operation**, at an honest node that generates it, in the *sender* DRP Object;
2. materializes into an operation, called **destination operation** at an honest node in the *receiver* DRP Object.

The signaling mechanism provides the *atomicity guarantee*: these two operations, across the sender and receiver objects, either get created together, or they don't get created at all. In this way, DRP signaling operates more like Inter-Blockchain Communication (IBC), ensuring synchronized state updates across distributed objects, rather than the traditional API model that relies on asynchronous request-response interactions.

In the current design, the following steps are involved for node A in DRP X to send a signal to DRP Y:

1. A subscribes to Y to obtain its hash graph.
2. A cryptographically signs and broadcasts a message M to its peers in Y, denoted as P_Y. M's payload expresses A's intent to generate a vertex in X containing the source operation, as well as a vertex in Y containing the destination operation.
3. Each node in P_Y verifies M's signature. Each of them then subscribes to X to obtain its hash graph, in order to verify the causal dependencies of the two vertices contained in M, as well as A's permission for generating these vertices.
4. If no verification failed, each node in P_Y signs and send an acknowledgement message directly to A.
5. When A receives valid acknowledgement messages from everyone in P_Y, A acknowledges back to them. A also proceeds with adding the source vertex in its hash graph of X.
6. Nodes in P_Y, hearing from A, proceed with adding the destination vertex in their hash graphs of Y.

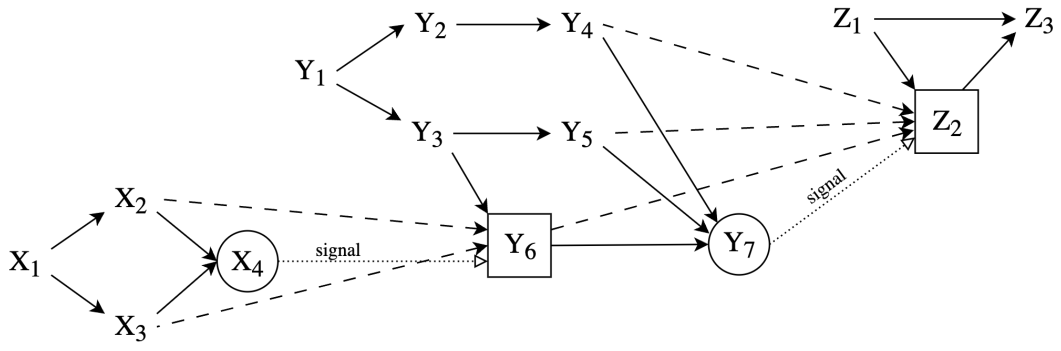

Figure 4: Hash graphs of DRP Objects X, Y, Z interacting by signaling (initial states are omitted). Signals are denoted as dotted arrows. Cross-object dependencies are denoted as dashed pointers. The vertices enclosed in circles are where signals are originated from. The vertices enclosed in squares are where the signals materialize into. In this example, the vertex pair (X_4, Y_6) is atomically generated by a signal, and the same for (Y_7, 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.

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

5 Invariants

Besides composability, preservation of invariants of all kinds is crucial for DRP Objects 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 [12]. 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 [13]. 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.

In the current design, the developer of a DRP Object is given the option to use the invariance preservation mechanics provided by the protocol. To use it, the developer specifies a set of invariants, along with a set of fixer functions to repair the violation of each of them by generating compensating operations. Upon DRP state changes, the new state is automatically checked against the provided invariants. At invariant violation, the associated fixer function is invoked to repair it.

6 Examples

In this section, we provide a basic example to illustrate the expressivity of DRP Object: a *social graph*.

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.

Algorithm 1 GMapGSet

```
state:
    $map$ : Map<any ⇒ Set>

function $Add\,(k : \text{any}, v : \text{any})$
    $if\ k \notin map \Rightarrow map.insert(k, \{v\})$
    $else \Rightarrow map[k].add(v)$

function $Get\,(k : \text{any}) \rightarrow$ Set<any>
    return $map[k]$
```

We can build a basic social graph with this blueprint. Algorithm 2 shows the pseudocode of a DRP Object named SocialGraph, which uses:

1. a grow-only set (GSet) to store user public keys;
2. a two-phase set (2PSet) to store posts;
3. a GMapGSet (from Algorithm 1) to store the timestamps when users are added.

7 Interaction methods

DRP defines a set of methods, or *verbs*, for nodes to interact with a given DRP Object. At the core of the protocol is the PubSub model for nodes in P2P networks to subscribe to DRP Objects and publish updates on them asynchronously. DRP Objects are identified as PubSub groups or topics. Nodes only subscribe to DRP Objects they are interested in. This keeps DRP Objects loosely coupled and helps reduce resource requirements for operating nodes. In contrast, to update the state

Algorithm 2 SocialGraph

```
state:
    $Users$ : GSet<User>                                   ▷ GSet is a blueprint
    $Posts$ : 2PSet<Post>                                  ▷ 2PSet is a blueprint
    $UserTimestamps$ : GMapGSet                            ▷ GMapGSet is from Algorithm 1

function $AddUser$ ($userPublicKey$ : User, $timestamp$ : Timestamp )
    $Users.Add(userPublicKey)$
    $UserTimestamps.Add(userPublicKey, timestamp)$

function $AddPost$ ($post$ : Post)
    $Posts.Add(post)$

function $RemovePost$ ($post$ : Post)
    $Posts.Remove(post)$

function $GetUsers$ () → Set<User>
    return $Users.Set$

function $GetPosts$ () → Set<Post>
    return {
        $p, \forall p \in Posts.Set \wedge p.user \in Users.Set$
        $\wedge\, p.timestamp > UserTimestamps.Get(p.user).max()$
    }
                                       ▷ the post needs to be (1) present in the Posts set,
                                       ▷ (2) added by a recognized user after the user was added
```

of any smart contract on a typical blockchain, participants must synchronize and process all contract activity across the network. Figure 5 illustrates the interaction pattern of DRP objects.

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

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

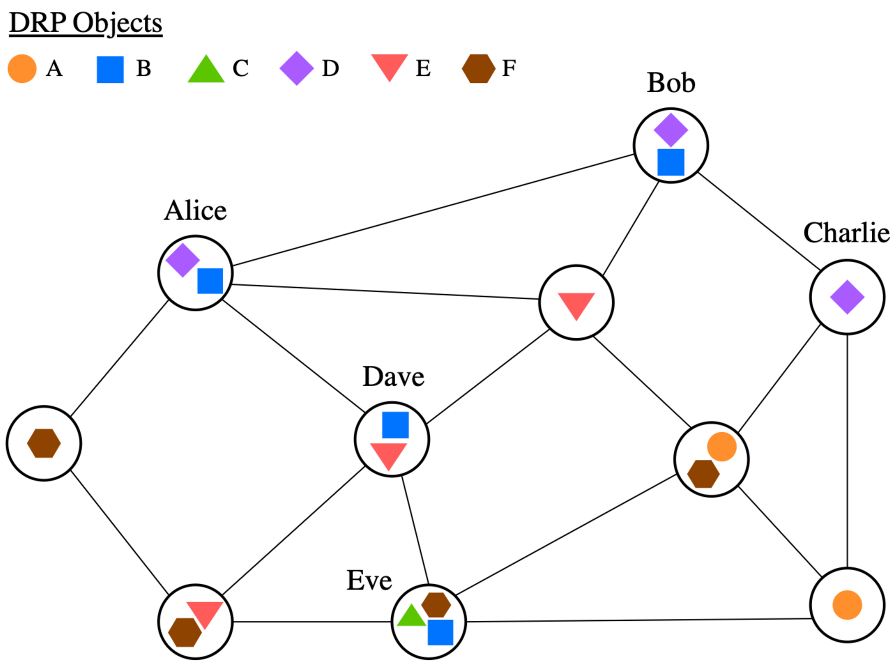

Figure 5: Alice, Bob, and Charlie are subscribed to DRP Object D. Alice, Bob, Dave, and Eve are subscribed to DRP Object B. All nodes are connected peer-to-peer.

8 Operation life cycle

The life cycle of a DRP Object operation differs significantly from a blockchain transaction.

- Every node that participates in a DRP Object keeps a copy of its hash graph locally.
- When a node performs a write (update) on the object, it generates an operation, which is added to the hash graph immediately.
- A node computes the DRP Object state by linearizing its local hash graph and applying the operations in that order.
- If a node performs a write and a subsequent read immediately, the read is guaranteed to observe the write.

In other words, DRP Object provides high responsiveness, where local operations are instantly effective [14]. In contrast, 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.

9 Access control

Access control is important for multiplayer applications. Common access control models include attribute-based [15] and role-based ones [16]. It gets tricky when a DRP Object's access control logic is specified within itself or another DRP Object, 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 [17], 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.

In the current design, Access Control Lists (ACL) are implemented by default for every DRP Object. Each of these ACLs have 3 different groups:

1. *Admins*: the set of peers that are able to grant and revoke the permissions of any other non-Admin peers. An Admin must specify the peer ID and the public key that identify a peer to generate an operation that changes its permission. The public key can either be a Ed25519 key, used for signing vertices upon generation, or a BLS key, used for signature aggregation to decide on finality.
2. *Finality Signers*: the set of peers that can decide on the finality of the vertices in the DRP Object's hash graph.
3. *Writers*: the set of peers that are allowed to generate operations to update the DRP state. This set is optional - for a permissionless DRP Object, this set is not used, allowing anyone to write to the object. For a permissioned DRP Object, only the operations generated by peers in the Writers set are accepted. The developer of a DRP Object controls its permissionless-ness by setting a flag.

10 Finality

A vertex in an object's hash graph is **final** if it is seen and validated by a threshold amount of participants in the DRP Object. This amount is specified by the developer of the object, analogous to a quorum in a voting system.

The ability to finalize the hash graphs of DRP Objects is beneficial in multiple ways:

1. External communication: the finalized parts of a hash graph can be transported to external systems for further use.
2. Snapshot: Snapshots of the object state can be derived from finalized parts (covered in Section 11).
3. Compaction: A node may consider pruning away vertices in the finalized parts (covered in Section 12).

In the current design, every node that receives a new vertex from the network signs it with their BLS key, and broadcasts the resulting signature. Nodes collect and aggregate these BLS signatures. For a given vertex, a BLS signature that represents at least a threshold amount of *valid* signatures over it is the proof that it is final. The validity of signatures depends on if the signer is among the Finality Signer set.

11 Snapshot

A DRP Object snapshot is an object state that is derived from a finalized part of the object's hash graph. It can act as a "state save" that may be worth persisted (*e.g.*, on IPFS [18] or any public blockchain of choice).

In the current design, when performing a snapshot, a special vertex is added on top of the frontier of a finalized part of the hash graph. This vertex carries no operation that mutates the object state. The object state computed from the perspective of this vertex is a snapshot. Anyone can verify the validity of this vertex's finality, which is equivalent to the validity of the corresponding snapshot.

12 Compaction

Grow-only hash graphs are problematic. Unbounded memory is needed, and large graphs take longer to bootstrap. Compaction is a mechanism to address this problem.

When a hash graph is compacted, an "old" portion of it gets pruned away. As this happens, a *new initial state* is obtained by applying the operations in that portion on the previous initial state.

In the current design, each node is allowed to compact the finalized portion of their hash graph. Specifically, at any moment, we call the frontier of this portion the *final frontier*. Vertices before the final frontier can be dropped. The final frontier moves forward within the hash graph as fast as the speed of finality. In this way, the memory requirement of participating in a DRP depends on the number of vertices that sit between the final frontier and the actual frontier.

One implication of compaction is that a vertex that is generated by an honest node may be dropped by other nodes, because the vertex's dependencies are already dropped by them during compaction, and thus become unrecognizable from their perspectives. Such a vertex becomes a *stale vertex*. This can happen when there are clusters of nodes forming a majority of the DRP's Finality Signer set, from which an honest node is distant. The crux of the problem lies in the need for the originating node of a stale vertex to learn about the staleness, such that it knows to generate the vertex anew. In the current design, each node regularly synchronize with a random peer. Through synchronization, a node gets to learn if any of its unfinalized vertex's dependencies have been dropped by its peers. It can then use such information to decide if the vertex is stale and requires regeneration.

Figure 6 illustrates the compaction process.

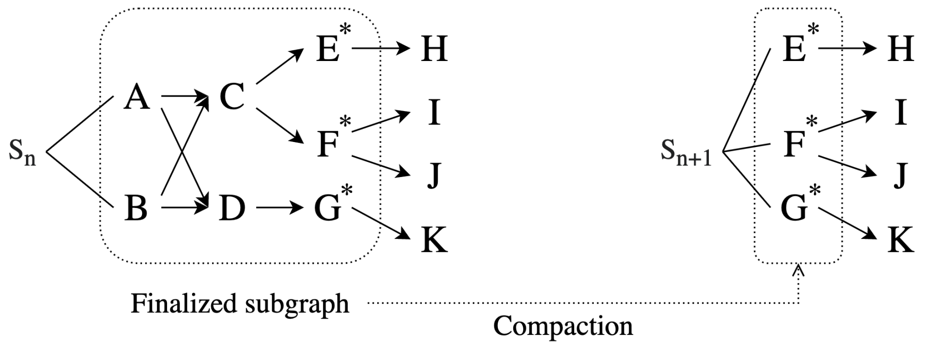

Figure 6: Compaction is performed on the finalized subgraph. Vertices on the final frontier are marked with asterisks $*$. Vertices that are before the final frontier (A, B, C, D) are pruned away.

Snapshots that are derived from compacted hash graphs are verifiable:

1. The hash graph of a DRP Object is routinely compacted.

2. When compaction happens, the new initial state forms a snapshot.
3. This snapshot carries with it a proof that attests to its validity, which comes in two parts:
   (a) compaction was performed correctly.
   (b) the finality of vertices involved in the snapshot was valid;

For part (a) of the proof (compaction correctness), we can leverage recursive zero-knowledge proof [19]. This may require the blueprint to be executable by a provable VM such as Cairo [20], RISC0 [21], or Valida [22]. The computation of the new initial state during compaction can be proven recursively (*i.e.* "I know a subgraph of valid vertices that contain the operations to forward the previous initial state to the new initial state"), yielding a recursive proof that sits alongside the snapshot. For part (b) of the proof (finality validity), we can verify that the BLS signatures collected on the frontier of the finalized subgraph are sufficient to finalize the subgraph.

13 Security

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

13.1 Hash graph pollution

Malicious actors may send invalid operations to pollute hash graphs. 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.

The validity of an operation depends on a number of factors:

- Signature is correct.
- The creator of the operation had the right permission.
- Applying the operation causes a mutation of the object state.

All these factors can be validated atomically. This means every honest node will reach the same conclusion on the validity of every operation.

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.

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.

The further study and mitigation of these security issues remain an active area of research.

14 Decentralized RAM

A compelling perspective on DRP is its ability to power a decentralized network that functions as a *decentralized RAM*. The following properties of the protocol support this perspective:

1. **Random access**. Users can synchronize on any DRP Objects of interest without having to synchronize every single DRP Object in the system, contrary to smart contracts on blockchains. This access pattern reduces the resource requirement of running a DRP node, making it more accessible for users to participate in the network.
2. **Closeness to compute**. We envision that most users of DRP Object-powered applications will operate nodes on their devices, holding object replicas locally. In other words, the protocol champions the principles of local-first software [6]. 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 RPC nodes, not unlike the common access pattern of cloud apps. This property is analogous to how random access memory sits closer to the CPU in the memory hierarchy.
3. **Ephemerality**. 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 DRP nodes and persistence providers (*e.g.* blockchains) is analogous to the data movement between RAM and disk storage, which is non-volatile.

Working with blockchains in concert, decentralized RAM has the potential to enable a new class of decentralized applications that can operate more dynamically and responsively than purely blockchain-based applications.

15 Conclusion

In this paper, we introduced DRP 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. DRP Object provides the programmability of common object systems and encapsulates coordination-free replication strategies. DRP specifies a set of interaction methods for DRP Objects 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 DRP Objects were discussed. Nodes that implement DRP keep replicas of DRP Objects locally, facilitate persistence without being primarily responsible for it, and disseminate DRP Object updates only to nodes that are interested in them. As such, a P2P network comprised of such nodes effectively functions as a decentralized RAM. Combining this RAM-like structure with blockchains, we reimagine the architecture of the world computer.

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 DRP Objects;
- Pierre Semanne and Goblin Oats, for advocating against a global finality system shared by all DRP Objects, 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] Jae Kwon. Tendermint: Consensus without Mining. Cornell University, 2014.

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

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

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

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

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. Attribute-based access control. *Computer*, 48(2):85–88, 2015.

[16] 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.

[17] 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.

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

[19] 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.

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

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

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