Topology (Draft 7)

Author: Thomas Hsueh (guiltygyoza), Jihoon Song, Kunho Kim

March 29, 2024

Abstract

Smart contracts have limitations in transaction costs, concurrency, and coupling. In this paper, we propose an alternative form of decentralized programs called live objects, or composable BFT-CRDT actors. Live object provides three key affordances: transactions are free by default, replicas progress at lock-free local speed, and cross-object interaction is loosely coupled. The key observation is that CRDTs are invariant confluent. They can achieve causal consistency while executing in a distributed manner with arbitrarily many Byzantine actors present, without relying on proof of resource or honest majority assumptions. Live objects can be used to implement multiplayer virtual environments, open social graphs, and many more to be imagined. We believe that the affordances provided by live objects complement those of smart contracts, and that combining them expands the design space of decentralized applications.

Acknowledgement

We appreciate the patience and feedback from Martin Kleppmann, Oskar Thorén, Anderson Chen, Lev Stambler, Joon Yun, Jay Oak, Matthew Demoy, Gregory Edison, Yu Yen, Martinet Lee, Ole Spjeldnæs, Justin Glibert, Zaki Manian, Alok Vasudev, Adam Goldberg, Colin Hong, Eric Wei, Peteris Erins, Wei Dai, Ventali Tan, Michael Gao, Ion Jo, Billy Rennekamp, and Ben Jay Amin.

1. Introduction

Most multi-user applications on the Internet today rely on centralized intermediaries to function. Multiplayer online games and social networks, for example, are typically architected as client-server applications. Servers make decisions authoritatively about what happened in the interaction among users, and maintain sensitive user information. Online commerce also largely operates on the client-server model. While this architecture has seen tremendous success, it has a number of problems. First, conflict of interest arises when centralized intermediaries can increase profits by favoring themselves or some favorable participants over others. Second, interoperability among these applications is limited, even with the availability of various interchange formats. This is because most of these intermediaries operate on the business model of building and protecting network effects. Third, requiring all interaction to be finalized by remote intermediaries creates a bottleneck in efficiency, as the growth in client-side compute power continues to outpace the improvement in network latency and bandwidth.

The Bitcoin blockchain was created to settle transactions between counterparties atomically without relying on third-party intermediaries. As a generalization of Bitcoin's design, the Ethereum blockchain is a replicated state machine that allows permissionless invocation of user-defined programs, which enables users to create alternate forms of digital tokens, exchanges, and financial derivatives. What is common among blockchains is the use of some Byzantine consensus mechanisms for equal participants to agree on the inclusion and ordering of events in the presence of Sybil actors.

Despite the affordance of trust-minimized transactions, reaching consensus on decentralized networks can be slow. To improve throughput, a large number of alternative designs have been proposed, many of which focus on minimizing coordination when possible. In particular, hash graph-based approaches allow nodes to agree on the same causal structure among past events by simply gossiping, not voting [Swirlds]. Nodes sharing the same hash graph can yield the same total ordering of events by applying the same deterministic function over the graph, which means eventually consistent total-ordering of events can be achieved also without voting. These events can carry transactions as their payload, allowing a hash graph to represent the transition function of a replicated state machine. Various improvements have been made [Aleph, DAG-rider, Narwhal & Tusk, Bullshark], which underlie some of the blockchain systems recently proposed [Aptos, Sui].

On the other hand, the modular thesis of Ethereum scaling advocates for settling the execution traces of multiple rollups, or application threads, on the state of a shared mother thread. The problem is that each application still runs in a single-threaded model of transaction ordering and inclusion. Advocates for verifiable computing introduces validity proving to further reduce footprints on the mother thread, and justifies the centralization over provers and transaction sequencers on the application layer. This creates a bottleneck in liveness and further complicates the already-complicated protocol stack. If the open metaverse were to operate on decentralized networks, a million participants each transacting at 60Hz would require a decentralized transactional medium of 60 million TPS at negligible costs for users [SIGGRAPH 2019], a daunting challenge in horizontal scaling. Finally, smart contracts interact in a strongly coupled manner, limiting the flexibility and development speed of interdependent programs by different developers. What is needed is a new kind of decentralized program that supports massive concurrency with very low to zero fees and loosely-coupled interactivity.

In this paper, we explore the intersection of the hash graph-based approach for event transport over open peer-to-peer networks, Conflict-free Replicated Data Type (CRDT), and the actor model abstraction [Actor model]. Our motivation is to provide new affordances to the developers of decentralized applications that benefit from Sybil-immunity, highly available transactions [HAT], and lock-free asynchrony in their interaction. We also explore a potential interplay between these new affordances and those of blockchains.

2. Hash graph

We begin our exploration with a model: a distributed system that uses a hash graph to encode causality among events [Swirlds, BEC]. A hash graph is a directed acyclic graph where each node is a message that contains an event generated by a process in the system, and each edge represents a causal dependency between events. Every process is single-threaded such that a process cannot generate concurrent events; a multi-threaded process is represented as multiple nodes in this model. Denote the set of root nodes of the hash graph as the graph's heads. The heads are nodes without successors. An important property of heads is that no pair of messages in this set is comparable: all messages in the heads are concurrent to one another. Upon producing a new event e, a process must wrap it into a message of the form $m = (e, \vec{d}, sig)$. where $\vec{d}$ contains the causal dependency of e, which must be set of hashes of the graph's heads at the time of event production; sig is the process's cryptographic signature over $(e, \vec{d})$. Each process in this system can independently produce events and update its local replica of the hash graph. Each process also engages in some anti-entropy procedure with other processes on a periodic basis to exchange missing messages until their heads match, effectively merging their graph replicas. Having matching heads implies having matching hash graphs. Events whose causal dependencies are unrecognizable by honest processes will not be added to their hash graphs.

To illustrate, consider a hypothetical system comprising process A and B, separated by a network delay of 30 ms. Both processes start from the same initial state, denoted by the null message $\emptyset$. Each process generates messages at 60 Hz and reconciles its hash graph with the other process as fast as the network conditions permit. Figure 1 below shows the hash graphs of process A immediately before and after A generates message A_5. Process A has not learned about the grayed out messages, B_4 and B_5. Heads are circled.

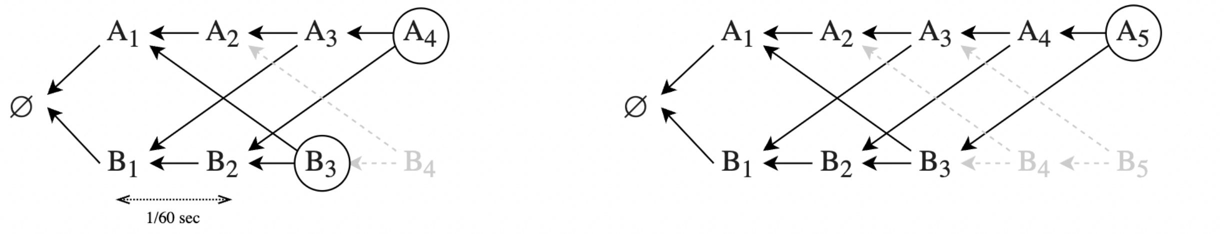
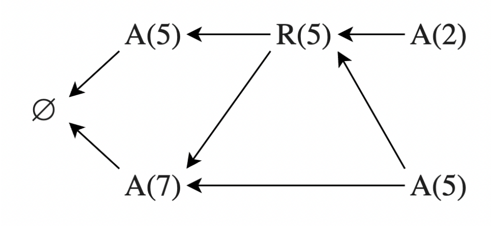


Fig. 1. The hash graphs of process A in a hypothetical system immediately before and after A generates message $A_5 = (e_{A_5}, \{H(A_4), H(B_3)\}, sig)$, where H is any collision-resistant hash function.

This setup is equivalent to a distributed program that recognizes a single transaction type "add(message)", has a merge function for merging replicas, with a single invariance "no loop is allowed in the graph". A loop means contradiction, since the causal relation is transitive. With the hash function being collision-resistant, creating a loop is not possible. Within the same system, merging two loop-free graph replicas yields a loop-free one. Thus, this distributed program is I-confluent [Coordination avoidance]. I-confluent programs can execute without coordination, preserving causal consistency, and with Sybil immunity: programs behave correctly even in the presence of arbitrarily many Byzantine processes [BEC].

3. CRDT

We can use the hash graph setup above to transport the updates of a CRDT [Merkle-CRDT; BFT-CRDT; Blocklace], making it a BFT-CRDT. CRDTs are abstract data types that abstract coordination-free replication strategies [CRDT]. CRDTs are I-confluent (i.e. any two states in the same join-semilattice merge into another state in the same lattice) and therefore can achieve causal consistency in the presence of arbitrarily many Sybil actors. We can call CRDTs “multiplayer data types”. Take operation-based CRDT as an example, whose specification provides semantics to resolve concurrent operations. To transport an operation-based CRDT with a hash graph, let the event payload of a message be an operation applied by a process. Figure 2 below shows a hash graph transporting some operations applied to a two-phase set CRDT. Since concurrent operations commute, applying the operations in any topological sort sequence of a hash graph yields the same outcome.



$$\begin{tikzcd}
 & A(5) \arrow[dl] & R(5) \arrow[l] \arrow[ddl] & A(2) \arrow[l] \\
\varnothing & & & \\
 & A(7) \arrow[ul] & & A(5) \arrow[ll] \arrow[uul]
\end{tikzcd}$$

Fig. 2. A hash graph that transports some operations applied to a two-phase set CRDT. $A(x)$ denotes an operation that adds element x to the set. $R(x)$ denotes an operation that removes element x from the set. Starting from an empty set, the resulting set after applying the operations in any topological sort sequence of this hash graph is $\{7, 2\}$.

The commutativity of concurrent operations allows a hash graph-transported CRDT to execute fairly and consistently without needing to decide on a single total-ordering of messages. This stands in contrast to DAG-based consensus algorithms [DAG-rider; Bullshark; Tusk] which uses shared random coin to total-order transactions fairly. The total-ordering requirement comes from the fact that Turing-complete smart contracts are not I-confluent programs.

Alternatively, the hash graph can transport delta mutations or delta groups for a delta-based CRDT [Delta CRDT]. Deltas are flexible: they can be merged into delta-groups, causally merge with state replicas, and algorithms exist to find the optimal small deltas [Join decomposition]. Delta-group provides an approach for batching locally-generated updates in the anti-entropy process for saving bandwidth.

4. Live object

We propose *live objects*, or composable BFT-CRDT actors [Actor model], as a new class of decentralized program. Live objects have the following three affordances:

1. *Zero-fee*: Live object achieves Sybil immunity without relying on proof of resource or majority stake assumption. There is no need to pay fees to a network of CPU-voting or staking-validator nodes for transaction inclusion and ordering. Therefore, interaction with live objects incurs no fees by default: end users interacting with specific live objects bear the computational cost of their own interaction. Possibility and countermeasures against grieving are discussed in Section 11.

2. *Lock-free*: Live object replicas can make progress independently at local speed. Without compaction, all transactions committed locally at a honest replica will be delivered and incorporated into all other honest replicas eventually with causal relations perfectly preserved. The complexities introduced by compaction are discussed in following sections.

3. *Loose coupling*: Each live object can be accessed independently. For a typical programmable blockchain, a user that wishes to transact with a smart contract would either run a full node locally or rely on centralized RPC providers who operate full nodes for profits. A full node downloads the full blockchain, whose state contains the entire "universe" of smart contracts. Most users shy away from the requirement for running a full node, more so when they are only interested in interacting with a small subset of smart contracts. In contrast, live objects can be subscribed to individually with gossipsub. A user machine runs the replicas of only the live objects of interest, with the exception of cross-object signaling to be discussed in Section 8. Additionally, live objects interact by asynchronous message-passing, in contrast to smart contracts, which interact by synchronous function calls.

Live object replicas exchange messages over the network, and execute the operations carried by those messages locally. This execution pattern allows live objects to "bet on the right side of the latency wall," leveraging the growth in client-side compute that outpaces improvement in network conditions.

Each live object is an instance of a *blueprint*, which is a specification of a compound CRDT. Blueprints are composable: a blueprint can be made from composing simple CRDTs as well as other blueprints. Blueprint correctness is equivalent to CRDT correctness, which relies on its state-mutating methods being monotonic, its merge function being order-independent, and its view function materializing its state with suitable concurrency semantics.

Live objects interact with the outside world by *signaling*, a live object-level term to differentiate from messaging at the causal broadcast layer. A blueprint specifies its *interface*, the set of signal types it handles.

At any time, a live object takes one of two forms: *active* or *dormant:* active form has at least one replica generating and accepting updates, while dormant form is a snapshot. A live object can be brought up from dormant to active, and be ejected from active into dormant. We consider IPFS as the default protocol for keeping dormant live objects [IPFS]. Live object replicas can read blockchains; writing to blockchains requires suitable global snapshot mechanisms, discussed in the following sections.

Below is the pseudocode of a blueprint implementing a primitive one-dimensional multiplayer space on integer coordinates with collision resolution. Collision resolution is achieved by endowing each player with a "repulsive field". A caveat of this approach is that tunneling occurs if replicas merge late: application-level causality is correlated with prevailing network conditions.

Algorithm 1 1D multiplayer space as state-based CRDT

```
state:
    Map{$\mathbb{I} \to \mathbb{N}$ <PNCounterCRDT>} : $positions$

constant:
    Integer : $F$                                                    ▷ strength of the repulsive field

interface:
    $Move\ (i, isRight, signature)$

on signal $Move$
    verify $signature$ against player $i$'s public key
    mutate $positions[i] \leftarrow positions[i] + (isRight\ \ ?\ 1 : -1)$

function MaterializedView
    return $positions$
end function

function Merge($positions_1, positions_2$)
    $delta = \{\}$
    for $i \in \mathbb{I}$ do
        for $positions \in (positions_1, positions_2)$ do
            for $d \in (-1, 0, 1)$ do
                $delta[positions[i] + d]$ upsert $\{i : d \times F\}$
            end for
        end for
    end for
    $positionsAverage$ =Map$\{(i : \lfloor (positions_1[i] + positions_2[i])/2 \rfloor) |\ i \in \mathbb{I}\ \}$
    return Map{(
         $i : positionsAverage[i]$
        $+\ deltas[positionsAverage[i] - 1].filter((k, v) \Rightarrow k \neq i).sum()$
        $+\ deltas[positionsAverage[i]].filter((k, v) \Rightarrow k \neq i).sum()$
        $+\ deltas[positionsAverage[i] + 1].filter((k, v) \Rightarrow k \neq i).sum()$
        $)| i \in \mathbb{I}$
    }
end function
```

Below is the pseudocode of a blueprint implementing a simplistic open-access social graph [Farcaster].

Algorithm 2 Social graph as state-based CRDT

```
state:
    GSetCRDT<User> : $Users$
    TwopSetCRDT<Post> : $Posts$
    TwopSetCRDT<Like> : $Likes$

interface:
    $AddUser$
    $AddPost$
    $RemovePost$
    $AddLike$
    $RemoveLike$

on signal $AddUser$
    verify $signature$
    mutate $Users.add(userPublicKey)$

on signal $AddPost[RemovePost]$
    verify $signature$
    mutate $Posts.add[remove](post)$

on signal $Like[Unlike]$
    verify $signature$
    mutate $Likes.add[remove](like)$

function MaterializedView
    $activePosts \leftarrow \{p \wedge (p \in Posts.AddSet) \wedge (p \notin Posts.RemoveSet)\}$
    $activeLikes \leftarrow \{l \wedge (l \in Likes.AddSet) \wedge (l \notin Likes.RemoveSet) \wedge (\exists p \in activePosts, p.postHash \equiv l.postHash)\}$
    return $\{activePosts, activeLikes\}$
end function

function Merge($\{Users1, Posts1, Likes1\}, \{Users2, Posts2, Likes2\}$)
    return {
        $Users1 \cup Users2$,
        $Posts1 \cup Posts2$,                                        ▷ union operator is defined for TwopSet
        $Likes1 \cup Likes2$,
    }
end function
```

5. Global snapshot

A global snapshot of a live object is the merging of hash graph replicas from all honest nodes. There are many motivations for performing a global snapshot. For example, a live object may "commit" its state on a blockchain by generating a set of transactions onchain. Since an onchain transaction is considered irreversible, it is important to incorporate hash graph replicas of a running live object from as many honest nodes as possible.

A naive approach is to silence the system, collect all replicas, discard dishonest ones, and merge the rest into one. Since all nodes of a hash graph specify their causal dependencies within the graph, a correct hash graph is always a consistent cut in the space-time diagram of a live object.

There are four problems with the above approach. First, a dishonest node performing the procedure can include incorrect replicas in the snapshot. Second, it may not be desirable or possible to silence a live object for the purpose of snapshotting. Third, given network conditions and dynamic membership, it may be impractical to hear back from all honest nodes before finalizing a snapshot. Finally, long-running live objects can have large hash graphs, making the transport of entire replicas bandwidth-consuming.

To address the first three problems, we can introduce a decentralized pacemaker to reach threshold-based asynchronous consensus on which set of hash graph replicas actually make it into a snapshot. Some live objects may prefer to be snapshotted periodically at predefined rates. Some may prefer on-demand snapshotting instead of periodic ones. Section 9 discusses how these preferences can be accommodated.

To address the final problem, we can leverage the fact that the heads of a hash graph effectively summarize the entire graph by *containing* all preceding nodes. Voting on heads is equivalent to voting on whole graphs. Another way to save bandwidth in transporting hash graphs is to perform compaction, to be discussed next.

6. Compaction

Grow-only hash graphs are impractical: unbounded memory is needed, and large graphs take longer to bootstrap for new nodes joining a live object. Compaction, or garbage collection, serializes the state of a CRDT by evaluating the operations (or deltas) carried by a portion of the hash graph before pruning that portion away. Compaction discards causal information. Compaction is perfectly safe only when performed over stable nodes. The following shows a hash graph before and after a safe compaction with node F and G confirmed to be stable: all future messages will have happened-after F or G. To support compaction, a live object needs to hold its compacted state beside its hash graph:

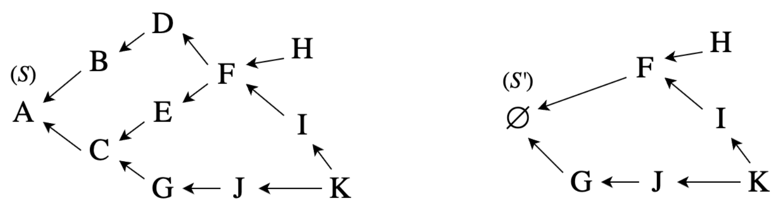


Fig. 3. A hash graph before and after compaction, where F and G are stable. $\emptyset$ denotes a message with null as payload, while S and S' denote states. S' is obtained from compacting messages A, B, D, C, E.

Perfectly safe compaction can be impractical: message delivery is eventual and unbounded, and replica membership can be dynamic. Compacting before confirming stability is unsafe. The implication of unsafe compaction is that new messages may arrive with causal dependencies no longer recognizable. Both Tusk and Bullshark choose to re-inject transactions to a later pace-maker round to preserve some notion of fairness (also called the validity property of reliable broadcast: transactions generated by correct nodes must be delivered to all other correct nodes eventually). However, re-injected transactions will not have the original causal dependencies. It is impossible to preserve fairness, causality, and have bounded memory at the same time. The decision on how to make this trade-off should be left to the developers of live objects: some live objects may prefer user-activated compaction to force stability on the basis of user consent, while other live objects may prefer to be compacted at the moment they are snapshotted. Section 9 discusses how these preferences can be accommodated.

7. “RAM”

We envision live objects to serve as the “random-access memory” of the world computer [Ethereum], an analogy that is based on three observations:

1. *Random access*: Users can access any live objects of interest without having to synchronize all live objects up to their latest states, contrary to smart contracts. This access pattern reduces the memory requirement of running a node. We expect that most users will be able to run local replicas of the live objects they access, contrary to blockchain users, who primarily access blockchains via centralized RPC nodes.
2. *Closeness to compute*: Live object replicas reside on user machines and afford users local transaction speed. We analogize these characteristics to the role RAM serves in a CPU.
3. *Ephemerality*: A live object exists in dormant snapshot form on IPFS when no user is accessing it. This transition pattern between active and dormant forms can be analogized to data movement between RAM and main memory.

Live objects offer a different set of affordances from smart contracts. We believe combining live objects and smart contracts expands the design space of decentralized applications.

8. Interaction

Live objects interact by sending signals asynchronously. A live object's signal handler method may request to signal another live object identified in a global namespace. Every cross-object signal (COS) originates from an externally-generated signal. Terminology-wise, we say a *sender* object has a *COS-initiating* message that produces a *COS-delivered* message at a *receiver* object. COS must specify its causal dependencies both with respect to the hash graph of the sender object and that of the receiver object; in particular, the causal dependencies of the COS-initiating message must be included.

To accomplish this and ensure delivery, the client that intends to initiate a COS must operate a replica of the sender object and a replica of the receiver object. Interactions can be shown diagrammatically among hash graphs, where a COS "glues" together its initiating message and its delivered message.

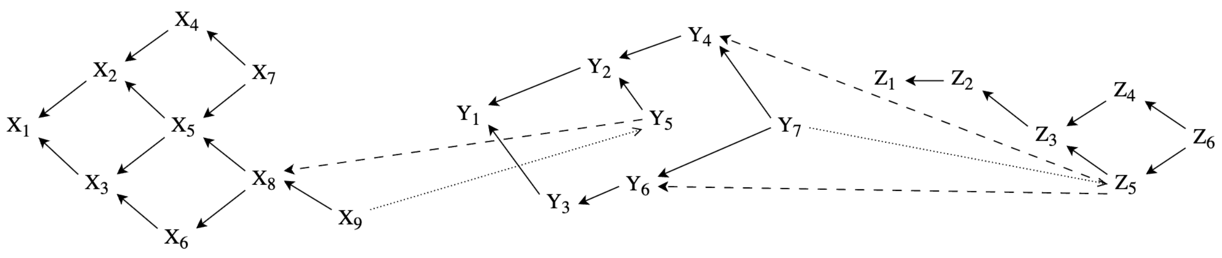


Fig. 4. Hash graphs of live objects X, Y, Z interacting by signal-passing. Cross-object signals are denoted as dotted arrows. Cross-object causal dependencies are denoted as dashed pointers. In this example, the COS-initiating message X_9 (produced by an externally-generated signal) produces the COS-delivered message Y_5. Same with messages Y_7 and Z_5. The causal dependencies of Y_5 is $\{X_8, Y_2\}$. The causal dependencies of Z_5 is $\{Y_4, Y_6, Z_3\}$.

9. Threshold logical clocks

This section provides an intuition using threshold logical clocks for pace-making and pruning hash graphs [TLC].

Global snapshot is necessary for "gear-shifting" from the lock-free asynchrony of live objects to synchronous systems such as blockchains, while compaction is necessary for avoiding unbounded memory. For permissioned live objects, replicas may be given equal rights to manually snapshot, compact, and save the compacted snapshots to IPFS. A permissioned live object may also designate a leader to perform global snapshots using suitable algorithms such as [Chandy & Lamport].

For other live objects, secure auto-saves and auto-pruning will be preferable. We expect the following common case demand from open-access live objects: on a periodic basis, perform a "fair" global snapshot of its hash graph replicas, compact all messages up to the heads of this snapshot, and save this compacted snapshot to IPFS. Given network conditions, correct replicas can differ drastically; malicious replicas may fall behind intentionally or inject messages with false causal dependencies. Deciding what collection of replicas to merge into a snapshot, in the presence of malicious actors, requires decentralized consensus.

We can use a leader-less proof of stake asynchronous consensus mechanism, where a fair snapshot is defined as the merging of replicas backed by at least $2f + 1$ out of $3f + 1$ total stake. We can adopt the layered architecture tradition and implement this consensus mechanism on top of threshold logical clocks as pace-makers.

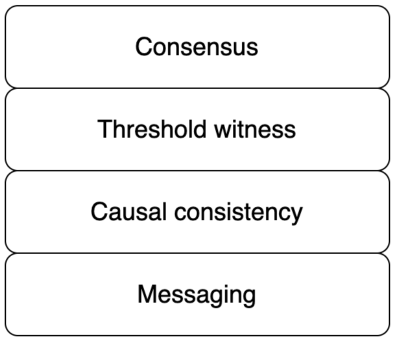


Fig. 5. A layered architecture, adapted from [TLC].

First, construct a messaging layer using suitable point-to-point transmission protocols. On top of this layer, construct a causal consistency layer where each participating node constructs a hash graph to record messages and the causal relations among them, as introduced in Section 2. Combining these two layers allows the construction of live objects. On top the causal consistency layer, construct a threshold witness layer that accepts arbitrary payload from upper layers and perform the threshold-witness procedure. On its top, construct a proof of stake consensus layer. We skip the randomness layer described in [TLC] because for CRDT programs, concurrent updates commute, fair total-ordering is not needed to reach consistency.

Consider a live object X that desires to be periodically pruned. For a proof-of-stake protocol to guarantee this operation, construct a network of protocol nodes that each subscribe to live object X. In this way, protocol nodes receive updates about X, and perform anti-entropy between themselves periodically. Protocol nodes are responsible for virally advancing the threshold logical time asynchronously. Each participant of live object X maintains connection with some protocol nodes to stay up to date of their local logical time, and to include this time in every message sent $m = (e, t, \vec{d}, sig)$. Consider a protocol node N whose local logical time just advanced to t. As soon as this happens, node N attempts to advance to $t + 1$ by performing the following steps:

1. Broadcasts the heads of its X graph replica to all other protocol nodes as $m_t^N = (\vec{h}_t^N, sig)$.
2. When another protocol node N_i receives m_t^N, it first verifies the signature, then merges the payload heads to its local graph as long as causal dependencies are recognized. It also produces a signature over the hash digest of m_t^N returns it along with its new heads as an acknowledgement message $Ack_i = (sign(m_t^N), \vec{h}_t^{N_i})$.
3. For incoming Acks, node N verifies their signatures, and merge the payload heads if their dependencies are recognized. For each Ack_i, N then produces a signature over its hash digest and broadcasts it as a witness message $Wit_i = (sign(Ack_i))$. Notice that merging the heads carried in Acks, if performed atomically, yields an outcome that is order-dependent. To maximize its “learning”, node N can postpone merging these heads until the moment of time advancement, and batch-merge them.

4. When another protocol node N_j receives Wit_i, first it checks the signature, then it returns a signature over the message as acknowledgement witness $AckWit_i^j$.

Now, assume from a suitable blockchain we have the following information:

1. A mapping from each protocol node public key to its current stake amount.
2. A $stake\_threshold$, such as 2/3 of total stake.

With these, node N advances its local logical time from t to $t+1$ if the following conditions are met:

1. For each Wit_i, if the corresponding $AckWit_i^j$ collected from protocol nodes represent $\geq stake\_threshold$ of stake, consider this Wit_i as successful, and its corresponding Ack as threshold-witnessed.
2. The aggregate threshold-witnessed Ack come from protocol nodes representing $\geq stake\_threshold$ of stake.

Figure 6 below illustrates the hash graph at node N upon its local logical time advancement:

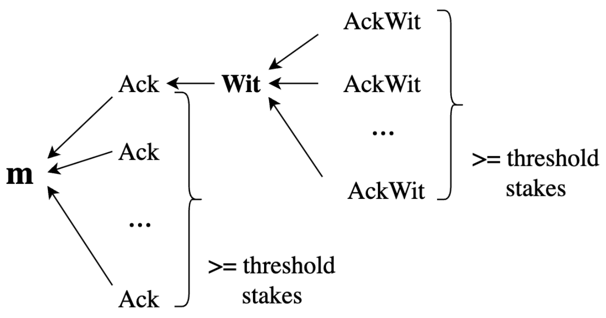


Fig. 6. Messages in bold are produced at protocol node N. All other messages are produced at other protocol nodes.

Following time advancement, node N performs pruning by compacting the nodes up to the heads of its local hash graph of X. The pruning step can be executed in parallel while node N begins its attempt to advance to $t+2$ by broadcasting m_{t+1}^N.

With periodic pruning, the rate of growth of a hash graph replica is bounded by the difference between the rate at which new messages are added to the graph and the rate at which pruning occurs. To minimize trust, we require each compaction run to be validity proven recursively over the previous compaction run. Notation-wise, we say for a protocol node N that just advanced to local logical time t, the tail of its hash graph is τ_t^N, which carries an empty event, the serialized state S_t, and a recursive proof PF_t representing the correctness of state advancement from initial state S_0 up to S_t. Denote the portion of the graph *before* a set of heads $\vec{h}$ as $G_{-\vec{h}}$, and the portion *after* $\vec{h}$ as $G_{+\vec{h}}$. When node N is ready to advance its time from t to $t+1$, denote its threshold-witnessed set of heads as $\vec{h}_{t+}^N$. The live object-specific compaction function takes S_n and $G_{-\vec{h}_{t+}^N}$ and produces S_{n+1}. Verify PF_t, then prove this verification and compaction together to produce PF_{t+1}.

The following diagram illustrates a node's replica right before a time advancement and after advancement from t to $t+1$:

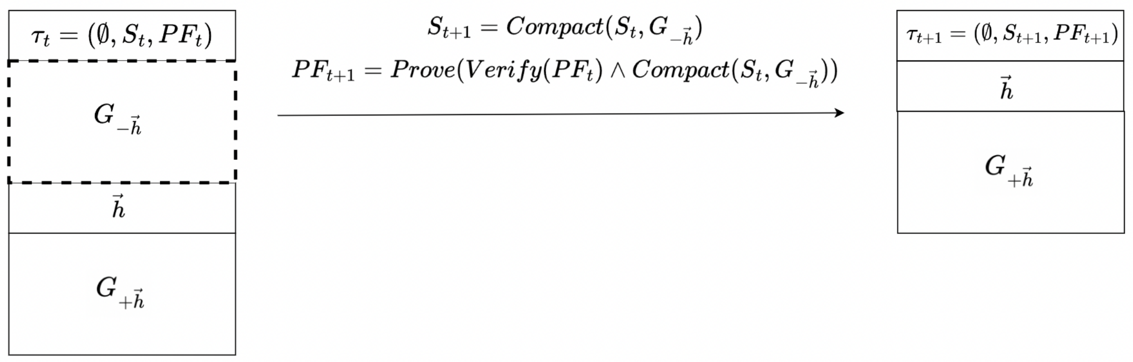

$\tau_t = (\emptyset, S_t, PF_t)$
$G_{-\vec{h}}$
$\vec{h}$
$G_{+\vec{h}}$
$S_{t+1} = Compact(S_t, G_{-\vec{h}})$
$PF_{t+1} = Prove(Verify(PF_t) \wedge Compact(S_t, G_{-\vec{h}}))$
$\tau_{t+1} = (\emptyset, S_{t+1}, PF_{t+1})$
$\vec{h}$
$G_{+\vec{h}}$

Fig. 7. The effect of compaction.

We can ask the same collection of capital-staking protocol nodes to perform time-keeping for any live object that needs it. To increase flexibility, we can allow each live object to decide when it needs the time-keeping and auto-pruning service from the protocol nodes to start and stop. Furthermore, some live objects will be dormant while other live objects are active, and dormant ones do not need time-keeping. This loose coupling between live objects helps reduce bandwidth requirements for operating protocol nodes.

A major problem with this approach is its communication complexity of $O(n^3)$. Another problem with this approach is that if a message timed t' arrives at a protocol node at local logical time t, and $t' < t$, the protocol node will not recognize the causal dependencies of the message. Similarly, if protocol nodes N and M have differing heads at time advancement t to $t+1$, they will prune away different sets of causal information, making their subsequent anti-entropy run problematic. Instead of rejecting causally-unrecognized messages, we can require the protocol node to force the dependencies of these messages to be the tail (message with null payload) of its local replica. This may be problematic, too: a message whose causal dependency is "forgotten" due to pruning may be indistinguishable from a message with incorrect causal dependencies in the first place. Finally, snapshots over successive logical time ticks should be monotonically increasing; potential violation in monotonicity and its implications need to be investigated.

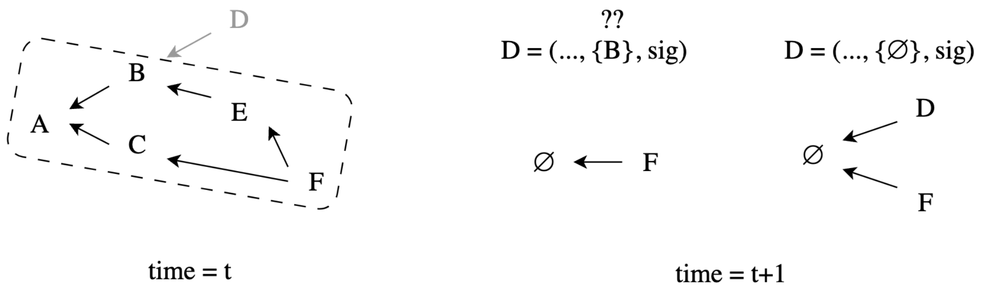


Fig. 8. A protocol node compacts message A, B, C, E when advancing its logical time from t to $t+1$, preventing it from understanding message D's causal dependency when it eventually arrives. D's dependency is forced to $\emptyset$ and included in the graph.

10. Privacy

End-to-end encryption is possible with live objects. Participants with a shared cryptographic key can encrypt the payload of all messages added to the hash graph. With end-to-end encryption, asking protocol nodes for auto-pruning service can be tricky because compaction requires evaluating the payload of messages. A solution is to have protocol nodes communicate threshold-witnessed but unpruned hash graphs at logical time advancement to user replicas, which perform compaction locally; protocol nodes can "dry-prune" the graph instead: the $\emptyset$ tail message in the dry-pruned graph would carry no state.

11. Security

Hash graphs can be polluted by messages with recognized causal dependencies but garbage payload. We can require a replica to type-check the update in a message payload before adding it to the graph. With auto-pruning activated, a garbage-filled hash graph can push the well-intentioned messages to be included in future logical time ticks, nullifying their causal dependencies.

Even without compaction, it may be possible for messages to report dependencies that are older than the replica's current heads without being detected. We can call this the *old-dependency attack*. To see how it works, consider an orderbook decentralized exchange as a live object. We require the following concurrency semantics:

1. Cancel orders take precedence over take orders.
2. When multiple take orders are mutually concurrent, without the presence of any cancel order, use a deterministic random function $dRandPick(Set)$ to pick a winner, potentially using some form of Fiat-Shamir heuristic.

At the programming level, assume the underlying hash graph provides the following API:

1. $\_graph.produceNode(Event)$: given a new event, generate a new hash graph node with correct causal dependencies

2. $\_graph.findTails(Set < Node >)$: given a set of nodes in the graph, find all predecessor-less nodes (tails). Intuitively, tails capture the set of the oldest events that are mutually concurrent.

Thus:

Algorithm 3 Orderbook decentralized exchange as operation-based CRDT

```
state:
    GSetCRDT<PostNode>: $PostNodes$
    GSetCRDT<CancelNode>: $CancelNodes$
    GSetCRDT<TakeNode>: $TakeNodes$

interface:
    $PostOrder$
    $CancelOrder$
    $TakeOrder$

on signal $PostOrder[CancelOrder; TakeOrder]$
    verify $signature$
    $node \leftarrow \_graph.produceNode(order)$
    mutate $PostNodes[CancelNodes; TakeNodes].add(node)$

function MaterializedView
    $takens \leftarrow \{\}$
    for $p \in PostNodes$ do
        $matchingCancels \leftarrow \{c \wedge (c \in CancelNodes) \wedge (c.orderHash \equiv p.orderHash)\}$
        $matchingTakes \leftarrow \{t \wedge (t \in TakeNodes) \wedge (t.orderHash \equiv p.orderHash)\}$
        $oldests \leftarrow \_graph.findTails(matchingCancels \cup matchingTakes)$
        if$(tails \equiv \emptyset) \vee (\nexists TakeNode \in oldests) \vee (\exists CancelNode \in oldests)$ continue
        $taken \leftarrow dRandPick(oldests)$
        $takens \leftarrow takens \cup \{taken\}$
    end for
end function

function Merge($\{PostNodes1, CancelNodes1, TakeNodes1\}, \{PostNodes2, CancelNodes2, TakeNodes2\}$)
    return {
        $PostNodes1 \cup PostNodes2$
        $CancelNodes1 \cup CancelNodes2$
        $TakeNodes1 \cup TakeNodes2$
    }
end function
```

With this live object, consider the following scenario:

1. Alice posts order #123.
2. Alice's replica exchanges heads with Bob's replica, allowing Alice to learn that Bob took order #123.
3. Alice "regrets" and performs an old-dependency attack by injecting a cancel of order #123 concurrent to Bob's take. This should not be allowed, given we require reporting the current heads as the correct causal dependencies of a new event.

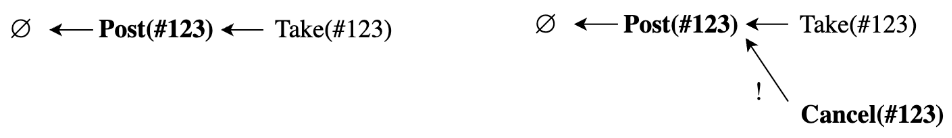


Fig. 9. Illustration of an old-dependency attack.

One possible countermeasure is for Bob's replica to remember having exchanged heads with Alice's replica, hence knowing Alice "should" know about the take order. The next time Alice and Bob's replicas exchange heads, Bob will detect Alice's amnesia. This would require adding a witness step to the anti-entropy procedure to let both counterparties know what each other knows. Even if Bob knows Alice injected a false concurrent event, how would Bob reliably report this misbehavior? How would others know if Bob is not wrongly accusing Alice? There might also be other configurations for an old-dependency attack. Finally, even if correct nodes agree on a valid accusation, how do we handle the downstream messages that have been generated and are causally depending on the old-dependency attack message? These problems require further investigation.

Conclusion

In this paper, we propose a new form of decentralized programs called live objects. It builds on top of the causal consistency model, using hash graph to verifiably transport and record causal relations among updates. Hash graph replicas can synchronize correctly, without coordination, even with arbitrarily many Byzantine actors present. CRDT abstracts these coordination-free causal replication strategies and allows for rich program composition and expressivity. Live objects are composable BFT-CRDT actors, interacting with each other by passing signals asynchronously. We imagine live objects to be snapshotted and stored on IPFS for persistence and permissionless retrieval. We discussed the necessity and complexity around global snapshots, compaction, and automatic pruning. We provided an intuition for constructing a proof of stake system to ensure the fairness of automatic pruning by requiring 2/3 stakes backing the witnesses of hash graph replicas before a protocol node merges them into one graph and compacts it to be used in the next local logical time tick. Finally, security problems on polluted has graphs and old-dependency attacks were discussed.

Reference

[Actor model] https://www.ijcai.org/Proceedings/73/Papers/027B.pdf

[Aleph] https://arxiv.org/pdf/1908.05156.pdf

[Aptos] https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf

[BEC] https://arxiv.org/pdf/2012.00472.pdf

[BFT-CRDT] https://martin.kleppmann.com/papers/bft-crdt-papoc22.pdf

[Blocklace] https://arxiv.org/pdf/2402.08068.pdf

[Bullshark] https://arxiv.org/pdf/2201.05677.pdf

[Chandy & Lamport] https://lamport.azurewebsites.net/pubs/chandy.pdf

[Coordination avoidance] https://www.vldb.org/pvldb/vol8/p185-bailis.pdf

[CRDT] https://pages.lip6.fr/Marc.Shapiro/papers/RR-7687.pdf

[DAG-Rider] https://arxiv.org/pdf/2102.08325.pdf

[Delta CRDT] https://arxiv.org/pdf/1410.2803.pdf

[Ethereum] https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[Farcaster] https://github.com/farcasterxyz/protocol

[HAT] https://www.vldb.org/pvldb/vol7/p181-bailis.pdf

[IPFS] https://arxiv.org/pdf/1407.3561.pdf

[Join decomposition] https://arxiv.org/pdf/1803.02750.pdf

[Live Distributed Objects] https://www.cs.cornell.edu/~krzys/krzys_dissertation.pdf

[Merkle-CRDT] https://research.protocol.ai/publications/merkle-crdts-merkle-dags-meet-crdts/psaras2020.pdf

[Narwhal & Tusk] https://arxiv.org/pdf/2105.11827.pdf

[SIGGRAPH 2019] https://blog.siggraph.org/2019/10/siggraph-spotlight-episode-30-tim-sweeney-and-the-metaverse.html/

[Sui] https://docs.sui.io/paper/sui.pdf

[Swirlds] https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

[TLC] https://arxiv.org/pdf/1907.07010.pdf