Directed Acylic Gossip Graph Enabling Replication (DAGGER) and Applications

v0.1.0

GenesysGo

September 29, 2023

Legal Disclaimer

Contents

The following list defines a few terms commonly used throughout this document. Some of these terms are adapted from SWIRLDS [1] and PARSEC [2].

1. **Directed Acyclic Graph (DAG)**: A directed graph is a collection of nodes with edges unidirectionally pointing from node to node. A directed graph is acyclic when no cycles are present when navigating any subset of edges from node to node (never visiting a node twice). In DAGGER, such a graph is used to record the history of gossip and derive a total consensus order.

2. **Operator**: A network participant helping to orchestrate consensus in the DAGGER system. In particular applications, like Shadow Drive, this participant take on additional tasks like storing data and auditing.

3. **Transaction**: A write request submitted by a user, auditor, or operator. A transaction can contain raw bytes, membership management requests, or Shadow transactions (Shadow Drive/Cloud actions e.g. store file, instantiate VM). The exact transaction contents depend on the DAGGER use case.

4. **Event**: A node in the DAG, which contains the hashes of its **Parent**, its **Self-Parent**, a timestamp, a payload (i.e. a **Block** or a **Consensus Payload**), and the creator's signature of the aforementioned. There are other auxiliary values held in memory such as booleans indicating whether the event is an **Agent** (and whether it is a **Shadowy Agent**), as well as the event's **Self-Index**, but these are variables that may be dynamic and need not be signed by the creator.

5. **Block**: A set of transactions that are packed into a Merkle Tree whose root hash is included in a node in the DAG.

6. **Bundle**: Upon the coalition of a Shadowy Council, which all operators will agree on, a group of events may finalize together and be consensus ordered. Such a group is called a bundle.

7. **State Hash**: Starting with some genesis hash as the state hash, a sha256 hash is repeatedly calculated from the previous state hash with the next consensus ordered transaction signature appended. This resulting hash is called the state hash and encodes the entire history of transactions applied to the genesis state.

8. **Bundlehash**: After executing the transactions contained in all of the events in a bundle, the state hash is recorded as the latest bundlehash. These bundlehashes are used as nonces in transactions to prevent reuse attacks.

9. **Consensus Payload**: A consensus payload can be thought of as a one-transaction block containing a membership management instruction, such as a request to join or leave the network. When it contains a join request, the event that contains this consensus payload is considered a genesis event. Consensus Payloads are not included in a block.

10. **Parent**: After an operator receives new events from a peer during a sync, the receiving peer shall create and add an event, containing an empty block if there is no transaction or consensus payload to include, to acknowledge this gossip. The gossip is acknowledged cryptographically by assigning the event's parent to be the latest event of a peer with which an operator has just synced.

11. **Self-Parent**: When an operator adds one of their own events to the DAG, its **self-parent** is assigned to be the previous event that an operator has made (none if genesis).

12. **Self-Index**: An integer assigned to an event indicating the event's index in the operator's timeline (i.e. the operator's 0th event, 1st event, etc.).

13. **Ancestor**: It is said an event A is an **ancestor** of B if one can reach the event A by recursively traveling along the event parents and self-parents of B.

14. **See (an event)**: It is said an event A can **see** an event B if B is an **ancestor** of A and B is not part of a **fork**.

15. **Strongly See (an event)**: It is said an event A can **strongly see** an event B if A can see events created by more than two-thirds of operators, each which can see B.

16. **Fork**: A fork is a collection of events that share the same self-parent. A fork can only come to be when an operator constructs two or more events that share a self-parent, either maliciously or accidentally. The consensus algorithm is robust to forks, finalizing exactly one or zero events in the fork.

17. **Level**: Every genesis event is assigned level 0. Then, every new event is assigned the maximum between the level of its parent and self-parent. If the event can **Strongly See** all **Agents** in this level, the level is incremented by one.

18. **Agent**: An event is an agent if it is a genesis event or if its level is greater than the maximum level between its parent and self-parent (increment case in **Level**). Essentially, an event is an agent if it is an operator's first event in a **Level**.

19. **Shadowy Agent**: An agent is a shadowy agent if there exists an agent at least two levels above the candidate agent which can strongly see a majority of agents one level below which all **vote** for the candidate agent.

20. **Shadowy Super Agent**: An agent is a shadowy super agent if it is the only shadowy agent created by the event's creator in that level.

21. **Vote/Cast**: It is said an agent A which is one level above a candidate agent C votes (yes/no) for C if it can see C. For agents that are more than one level above the candidate agent, it is said the higher level agent H votes for C if the majority of the agents H can strongly see one level below itself vote for C. Every 12th round is a **Rand Round**; if the voting agent's level is divisible by 12 and is not at the level just above the level of C, then it will either inherit the majority vote as before but only if the majority is a supermajority (more than two-thirds), or H will use the middle bit of its own signature as its vote.

22. **Shadowy Council**: An event's shadowy council is the first complete set[1] of shadowy super agents in a level that can see the event. Once such a shadowy council exists, the event can be consensus ordered.

23. **Rand Round**: Rand (or coin) rounds are used in SWIRLDS so that operators reach consensus eventually even if an attacker controls the flow of messages and splits votes. Rand rounds only occur for voting agents that are more than one level above the candidate agent. During a rand round, a voting agent will inherit the majority vote as in a regular round but only if the majority is a supermajority (more than two-thirds). Otherwise, it will use the middle bit of its own signature as its vote.

24. **Gatekeeper**[2]: A gatekeeper is defined as a participant with a cryptographic identity which grants users, operators and other use-case specific participants (like auditors) permission to join or use the network. Permission is granted via a cryptographic signature indicating some given input was verified, e.g. user payment for storage.

[1] In this context, a set of agents on some level comes before another set of agents on another level if their level is lower – the first such set is the set with the lowest level.

[2] Only relevant for a permissioned implementation.

1 Introduction

The world at large is becoming increasingly digital. For payments, storage, recreation, socialization, and much much more, we are becoming increasingly reliant on digital infrastructure. In their current form many of these technologies are opaque, and full of middlemen that introduce single points of failure and additional fees. Decentralized distributed consensus algorithms provides us with the opportunity to build robust, resilient, and transparent infrastructure for our digital world. Not only do these systems democratize the control over our shared digital infrastructure by removing central authorities and remove single points of failure, they also offer the opportunity to democratize access to the revenue streams generated by the digital infrastructure. This is no small matter. Over the last decade, the tech sector has grown to over a quarter of the S&P500, and the global cloud market is expected to surpass $4 trillion USD within a decade [3]. Furthermore, even for private infrastructure, Byzantine fault tolerant systems are more reliable.

This motivates a general purpose consensus mechanism that can be used to build infrastructure tailored to many use cases. This document outlines GenesysGo's *Directed Acyclic Gossip Graph Enabling Replication* (**DAGGER**), a scalable bandwidth-efficient distributed system for reaching consensus via an asynchronous graph-based Byzantine fault tolerant mechanism inspired by SWIRLDS [1] and PARSEC [2]. It also describes a particular application of the DAGGER system – Shadow Drive, a distributed data storage system – and how DAGGER can be used for other potential applications. GenesysGo's implementation is written in Rust but uses widely accepted standards and so can be implemented in any language.

2 DAGGER

A node in a DAGGER cluster is comprised of five components. The components, in the order an incoming transaction would see them throughout its life cycle, are as follows:

1. **Communications (Comms)**: This component is responsible for network I/O. It handles communication with peers in the network and clients that wish to submit/read transactions to/from the network.
2. **Processor**: This component has two closely related sub-components, which share a thread pool.
   (a) **Verifier**: This sub-module is responsible for verification and deduplication of incoming events from other peers during syncs.
   (b) **Forester**: This sub-module is responsible for verifying and packing new incoming transactions into Merkle trees, producing a block and a Merkle root hash for the graph module to include in an event; as well as verifying the Merkle root hash of the events received from syncs containing blocks.
3. **Graph**: This module is responsible for read/write operations on the directed acyclic graph (DAG), which include reaching consensus on transaction ordering.
4. **Controller**: This module is responsible for transaction execution after consensus ordering.

First, we describe how DAGGER reaches consensus. Then, the following sections delineate the responsibilities of each component toward reaching consensus. We provide some visualizations to aid in understanding the life cycle of a transaction, peer request, or client request in the system, and the interfaces between the components.

The DAGGER system reaches consensus on the ordering of transactions via computations on a directed acyclic graph held by every operator. The algorithm resembles the SWIRLDS [1] algorithm, but is modified to allow for dynamic membership rules, and the system performs only one-way syncs during gossip. It is asynchronous, leaderless, Byzantine fault tolerant, and highly bandwidth efficient. The bandwidth efficiency is gained from the consistency of the graphs across all operators coupled with the fact that each event in the graph serves as a vote for multiple blocks. That is, unlike other systems that must propagate blocks and **then** transmit $\mathcal{O}(N^2)$ or even $\mathcal{O}(N^3)$ messages to vote on and finalize each block, votes for a particular block are derived from the connectivity of the nodes in the graph and the contents of a small amount of metadata appended to the nodes. Votes are implicit – no explicit votes are transmitted over the network. This provides a significant advantage when compared to networks with a large number of operators such that the consensus bandwidth demand is comparable to the network-wide transaction bandwidth demand.

Before discussing how consensus is reached, we must understand how to construct the directed acyclic graph used to reach consensus. The graph can be understood as a collection of timelines – over time, every operator constructs a series of events. Events are comprised of several items:

- The signature of the event's **parent**, if it exists (ed25519 signature)
- The signature of the event's **self-parent**, if it exists (ed25519 signature)
- A payload, (e.g. a **Block** of transactions)
- A timestamp assigned by the operator (signed integer)
- The operator's identity (ed25519 public key)

A total order is rather obviously implied for the events in an operator's timeline, and for any transactions included in a block. We say that an event's **Self-parent** is the event that came before it in an operator's timeline. Every valid event produced by an operator has only one self-parent (except for their first genesis block) which is simply the event that came before it. Otherwise, there is a **Fork** that invalidates all future events created by an operator. However, the so-called **Parent** of an event is what encodes the history of peer-to-peer communication and helps reach a total order across all operator events. Using a gossip protocol, operators retrieve events from other operators that they do not currently have in their graph. After learning about new events from another operator [3], an operator creates a new event which assigns the latest event from the peer they synced with as the parent of the event. Figure 1 shows an example of such a DAG; the event E3, for example, denotes that operator E received events from operator B (in this case, B3, B2, B1, A2, and A1). These ever-growing eventually consistent directed acyclic graphs are the keystone of the consensus mechanism.

From some genesis state and hash, operations on this state and hash are orchestrated by repeatedly running the consensus algorithm on the graph, checking for new finalized transactions and executing them in consensus order. A transaction is finalized when the event it is included in has been consensus ordered. An event E is consensus ordered when there exists a level above E's level whose agents have all had their shadowiness assessed (see **Shadowy Event**) and which can all **see** E (the **Shadowy Council**). When it is consensus ordered, an event is assigned a consensus level[4], a consensus timestamp[5], and a whitened signature[6]; the events are in consensus ordered when they are sorted by consensus level, with ties broken by consensus timestamp, and ties broken again by whitened signature. This is the DAGGER system's ultimate goal, accomplished by facilitating peer-to-peer communication to exchange events (done by the communications component), verifying exchanged events and user transactions (done by the

[3]To minimize unnecessary computation and transmission, this step is skipped if an operator did not learn about any new events during a sync **AND** there is no block of transactions to include in an event.

[4]The first level above E for which all agents have had their shadowiness assessed, and which all see E.

[5]The median timestamp of the set of all

[6]A whitened signature is the signature obtained by XORing the event signature with each of the event signatures in the set of shadowy agents used in the previous step to determine consensus. This is the set of unique shadowy agents in the consensus level.

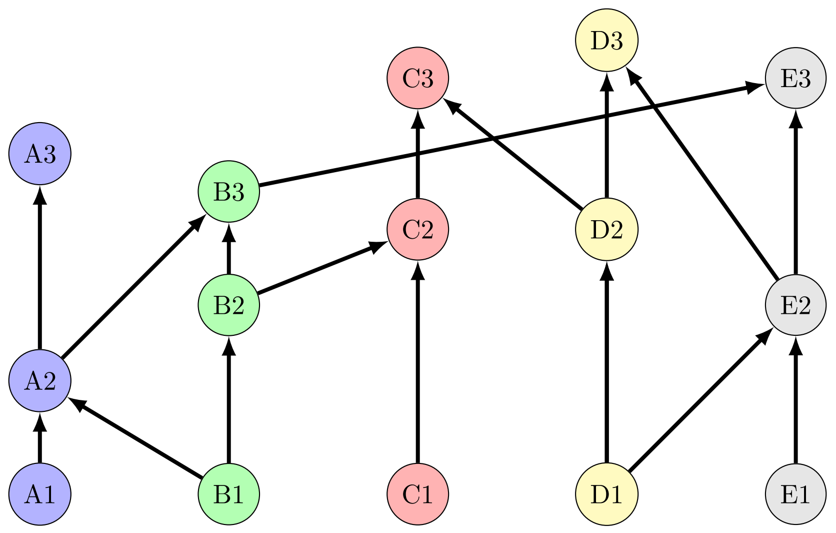

Figure 1: A directed acyclic graph recording the sync history of five different operators. Every column corresponds to the timeline of events for every operator. Each operator has a genesis event with no self-parent and no parent (A1, B1, . . .). After the genesis event, every event must have a self-parent but may not have a parent (A2 has a parent (B1) but D2 does not). Every event with a parent denotes that a sync occurred between the two peers. Because every event contains an operator's signature of its contents, which include the parent and self-parent signatures, this graph is cryptographically guaranteed to contain events created only by the specified creator with the parents that the creator specified. Tampering with the transactions in a block within an event or forging an entire event can be detected with a simple signature verification. It is this cryptographic guarantee that ensures all operators have eventually consistent graphs [1]. If two peers have the same event in their graph, all of the event's ancestors will be identical. Eventually consist graphs are mostly identical, varying near the top as operators add and propagate new events.

processor component), adding verified events and newly created events to the graph followed by running the consensus subroutines (done by the graph module), and then executing the transactions (done by the controller module).

2.1 Communications

The communications module initializes outgoing sync requests with peers, forwards sync responses from peers to the processor, handles incoming sync requests from peers, forwards transactions to the processor, and forwards RPC requests to the controller. Since the communications module is a message passing layer, we defer the specification of the messages passed and the operations performed with them to the appropriate layer which handles them. We do, however, briefly describe which messages are passed by communications module including when and how.

2.1.1 Outgoing Sync Requests

When requesting to sync with a peer to receive new events not yet in the local graph, a peer is randomly chosen from the current list of operators. Periodically, every 50th sync, the peer's lexicographic neighbor is chosen instead; peers are sorted by their public key and the next peer is chosen, with the final peer wrapping around to the first peer. This ensures that there is no time period in which a peer is left out of syncs for too long, reducing the tail latency of finalization. The sync response is forwarded to the processor module where it is verified before being digested by the graph module.

2.1.2 Incoming Sync Requests

An incoming sync request containing the peer's graph state summary is forwarded to the graph module. The communications module awaits a packaged sync response from the graph module containing all events that we have which the peer does not have, which is sent back to the peer.

2.1.3 Incoming Transactions

When a user submits a transaction, the communications module forwards it to the processor for verification. After verification, the transaction makes it way through the forester and the graph module which, upon inclusion in a block, sends back the signature for the event which contains the block in which the transaction was included. The communications module forwards this signature back to the user. Note that this does not mean the block has been finalized.

2.1.4 Incoming RPC Requests

When a user submits one of several possible RPC requests, whether it be to read a file or inquire about a recent block or transaction, it is forwarded to the controller. The controller sends back the result of the ledger query to the communications module, which forwards it to the user.

2.1.5 `dagger-p2p`

Incoming peer and client connections are handled asynchronously and concurrently with a peer-to-peer protocol `dagger-p2p` based on QUIC, written in Rust. The protocol provides a simple abstraction layer over QUIC which exposes a few functions: `connect_to_peer`, `send` and `receive`. Their functionality is self-explanatory, but we detail their arguments and internals. An endpoint object can be created, which exposes the `connect_to_peer` method. This method accepts a socket address, consisting of an IPv4 or IPv6 address and a port, and returns a connection to the peer conditional on a successful connection. This connection object has an inner

QUIC connection as defined in the QUIC specification, and exposes the `send` and `receive` methods. The `send` method accepts a reference to a byte buffer and asynchronously writes its length as a u32 and then its contents to a newly opened unidirectional QUIC stream with the peer. The `receive` method asynchronously accepts a unidirectional stream, reads the content length, and then reads the expected number of bytes in a message. Each stream thus encapsulates a message, and multiple messages can be sent across multiple streams. A message can be any set of bytes and can even be a subset of the message we would actually like to send, e.g. individual events while peers are syncing. Thus, `dagger-p2p` takes advantage of both of the primary benefits of QUIC – multiplexing and a lower latency handshake – while providing an exceptionally ergonomic API. Advanced users can provide a custom configuration for the endpoint to customize the maximum number of streams and internal buffer sizes.

2.2 Processor: Verifier & Forester

The verifier and forester are sibling modules responsible for verification of events, blocks, and transactions. There are several tasks performed at this stage: signature and input verification for transactions, signature verification for events, and root hash verification for blocks. When processing peer events, the verifier is responsible for the first two of these forms of verification, while the forester is responsible for verification of the root hash of the transaction Merkle trees associated with a block. When processing incoming user transactions, the verifier is again responsible for the first two of these forms of verification, while the forester gathers transactions to pack into a block and produces the Merkle root hash which represents the block. The verification process for events, blocks, and transactions vary. Verification of events and blocks are similar internal consensus processes. Transaction verification, however, varies in scope and complexity between and within DAGGER applications.

Transaction verification is comprised of up to three stages, all which can be done in parallel. The first stage is a simple ed25119 signature verification of the transaction payload, which verifies that a user intended to authorize this request. To prevent reuse attacks, the transaction payload signed by the user includes a nonce: a signature of one of the most recent 1000[7] **Bundlehashes**. Since DAGGER consensus is asynchronous, it may be unclear to a particular operator what another operator's most recent signatures are. As such, the transactions in a block must only contain the nonce signatures generated by the operator. At execution time, all operators have access to the recent history of bundlehashes and can verify the transaction nonce is equal to one of the last 1000 bundles since finalization. Duplicate transactions (with identical signatures) are either discarded at block inclusion time by honest operators, or ignored at execution time (if the block was constructed by another operator) with malice reported. This is the second stage (nonce + deduplication). The third stage is application specific, but will generally involve verification of data which is external to the system such as verification of an external smart contract interaction. As an example, when a user pays for storage on a DAGGER-based Shadow Drive on an L1 (e.g. Solana), the DAGGER system must verify that the payment for storage was made via the appropriate smart contract interaction. In a permissionless deployment of DAGGER, this could be done through a bridge, which itself is an application of DAGGER, or another such as a zkBridge [4]. In a permissioned deployment of DAGGER, only signatures by the set of gatekeepers indicating they've verified the transaction input are required[8].

[7]Subject to change

[8]It is possible for two operators to have different "current" sets of gatekeeper public keys if, for only one of the operators, a gatekeeper change transaction was recently finalized and executed. In this case, the operator with the outdated gatekeeper list will simply discard (or temporarily cache) this payload until the event containing the gatekeeper change is finalized and executed. This is an unlikely scenario, as the user would likely not use a gatekeeper whose joining has not yet been finalized, and although consensus is asynchronous it should not be delayed by much between nodes unless an operator has restarted their node and is catching up. Nevertheless, the user can simply resubmit their transaction.

Block verification refers to verifying that the Merkle root hash associated with a block another operator has constructed is indeed derived from a fanout $F = 2$ Merkle tree where the leaves are the signatures of the transactions in a block. This Merkle root hash encodes a unique transaction order, so this verification ensures that the order specified in the block was the same as that which was signed off by the operator who created the block. Merkle trees may suffer from second image attacks, since one can build equal Merkle trees where one node hash is interpreted as a leaf. However, by prepending 0x00 to leaves and 0x01 to nodes, we can prevent second image attacks. By requiring Merkle trees be as balanced as possible, we can have a deterministic tree topology given the number of leaves[9]. This enables better parallelism for faster verification. This block verification is done for an event only when the event payload is a block. For core consensus payloads, only the single transaction verification is done. Additionally, since core consensus payloads are transactions by the operator, the nonce used should be the signature of the operator's previous event.

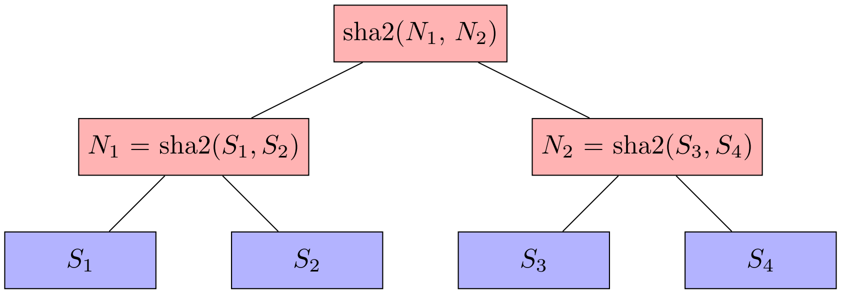

Figure 2: Transaction Merkle Tree whose root hash encodes a unique ordered set of transactions. Prepending 0x00 to leaves and 0x01 to nodes prior to hashing prevents second image attacks.

Event verification refers to performing an ed25519 signature verification on the event signature, which is an operator's signature of the following concatenated bytes:

- The event's parent signature, if present (ed25519 signature)
- The event's self-parent signature, if present (ed25519 signature)
- The Merkle tree root hash if a block is included, otherwise consensus payload signature.
- The UTC unix timestamp at event creation (signed 64-bit integer as little Endian bytes)
- The creator's ed25519 public key bytes

This verification helps uphold the guarantee of consistent graphs across all operators.

2.3 Graph

At a high level, the graph module has two core responsibilities: to digest sync responses to build up the directed acyclic graph, and to analyze the graph to derive the consensus ordering of events. There are also collaborative tasks that the graph component must perform so that the system and the cluster can function such as summarizing the graph state and gathering events to send to peers. In addition, there are several graph operations which are subtasks that are worth illustrating and specifying.

The graph module is the only module that has read and write access to the directed acyclic graph. As such, there are several checks the graph module must perform before inserting an

[9]Perhaps uninterestingly, this is also a second quickly verifiable defense against second image attacks where a tree is left very imbalanced as a result of replacement. It is uninteresting because all cases are covered by the first check which is necessary to carry out in all cases.

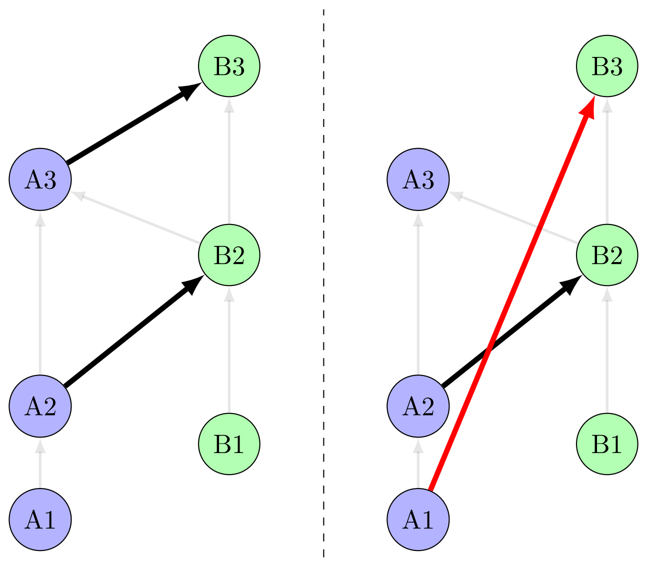

Figure 3: (Left) Event B3 has a parent whose timestamp comes before B3's self-parent's (B2's) parent's timestamp. (Right) Event B3 does **not** have a parent whose timestamp comes before B3's self-parent's (B2's) parent's timestamp. As in PARSEC [2], we call A1 a stale event in this context. Constructing and signing off on an event such as B3 which refers to a stale event as a parent is considered malicious.

event when digesting a sync response. The events contained in a sync response are digested in topological order[10]. Given an event in a sync response whose signature has been verified by the processor, the following checks must be performed on the event by the graph module prior to graph insertion:

- The time indicated by the event's timestamp must not be in the future[11], and must be after that of the timestamp of its parent (if present) and its self-parent (if present).
- The time indicated by the event's parent's timestamp must be after that of the event's self-parent's parent's timestamp (see Figure 3 for a visualization).
- The self-parent signature provided must indeed belong to the event's self-parent. That is, the indicated self-parent event must come just before the event in the operator's timeline. This verification implicitly verifies that an event's self-parent's creator is the event's creator. If the event's indicated self-parent comes before.

The timestamp verifications provide the guarantee that the graph's topological acyclicity implies temporal monotonicity for all events. In other words, all arrows are guaranteed to point from earlier events to later events.

[10]It is expected that a peer will send a sync response with events roughly in topological order. This is not strictly necessary, but is better for performance.

[11]If such an event were to be accepted in the graph, it would not interfere with consensus in any major way. Furthermore, an operator would not benefit from doing this as they would essentially lock themselves out of block/block/event production in all honest operator graphs due to the checked condition following this footnote regarding the parent and self-parent timestamp. Additionally, there is no straightforward benefit to an operator in assigning an timestamp to an event which is earlier than the present but later than their last event; the consensus timestamp assigned to an event is the median of the timestamps of all events in its **Shadowy Council**.

2.3.1 Collaborative Reads

The graph module reads the graph to prepare a state summary for outgoing sync requests, which uniquely summarizes the state of the graph. This state summary contains for each operator:

- The operator's ed25519 public key
- The operator's latest event signature (ed25519 signature)
- The latest event's index within the operator's timeline (e.g. this operator's 3rd event)

This state summary contains enough information for the syncing peer to detect forks and to retrieve events that the requesting peer does not have. The communications module receives the state summary upon its request, sends it to the peer, and awaits a sync response.

Additionally, the graph module reads the graph to prepare a sync response. Upon receiving a sync request including the syncing operator's graph state summary, the graph module must gather all events the syncing peer does not have. The communications module receives the sync response, and sends it to the peer.

2.4 Controller

The controller module performs reads from and writes to the ledger. This is where different use cases of DAGGER will vary the most. For filesystem applications of DAGGER (e.g. Shadow Drive), this includes operations like sharding and erasure coding. For other use cases like layer 1 blockchains, oracles, bridges, and VM orchestration (see §4), this is where their respective operations are executed.

Regardless of the application, the controller must still execute core DAGGER consensus payloads. This can include membership management such as handling operators joining, leaving, and updating the reputation of operators (e.g. uptime and malice), and updating stake if using Proof-of-Stake.

2.5 Visualizations

This seciton provides diagrams to help visualize the flow of data in the system. As previously mentioned, users submit transactions, which are received by the Communications module. Read requests (RPC Requests) are forwarded to the Controller module to read from the ledger, while write requests (Transactions) are sent over to the processor module for verification and packing.

2.5.1 RPC Requests

The lifetime of an RPC request can be visualized in Figure 4. The Communications module forwards transactions to the controller, which reads data from the operator's local ledger.

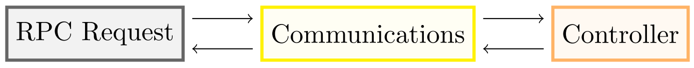

Figure 4: The lifetime of an RPC request. The Communications module forwards RPC requests to the controller. The controller performs reads from the current ledger state. The result of the read operation is sent back to the controller and forwarded to the user.

2.6 Transactions

The lifetime of a transaction can be visualized in Figure 5. The Communications module forwards transactions to the processor which verifies the transaction payloads, signatures, and packs transactions into Merkle trees. This stream of Merkle trees is made available to the graph module which inserts them into the DAG to be consensus ordered. Once it has been consensus ordered, it is sent over to the Controller for execution.

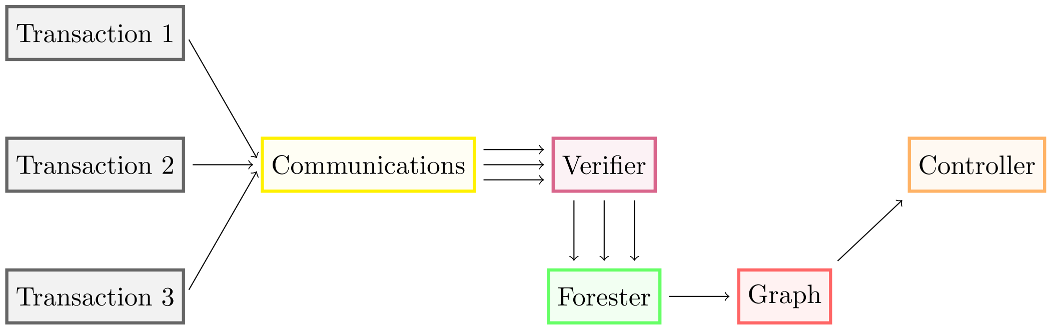

Figure 5: The lifetime of a transaction. Transactions pass through the system individually through the Communications module and are verified in the Processor module. Then, they are packed into merkle trees in the Forester module. Once packed, the trees are sent to the Graph module for inclusion in the DAG. After consensus ordering, it is sent to the Controller for execution.

2.7 Membership Management

One of the primary functions of a distributed consensus algorithm with dynamic membership is to keep track of who exactly is participating in the consensus. DAGGER has several core Consensus Payloads to aid in this function, which vary in the permissioned and permissionless implementations.

2.7.1 Entry & Exit

To join a permissioned DAGGER network, the operator submits a signed transaction containing an Entry consensus payload to a gatekeeper which verifies the operator is allowed to join and has staked appropriately on a supported blockchain. Upon signing, the gatekeeper submits the transaction to the network. After the transaction is finalized, the operator can construct a genesis event with level equal to zero, and the timestamp equal to the consensus timestamp of the finalization event.

To leave a permissioned DAGGER network, the operator submits a signed transaction containing an Exit consensus payload to a gatekeeper which verifies the operator has requested to exit on a supported blockchain. Upon signing, the gatekeeper submits the transaction to the network. After the transaction is finalized, an unstake voucher will be made available after 1000[12] levels, during which the operator is still expected to participate in consensus.

2.7.2 State

Every operator has some state on the ledger which records metadata including level at which they joined, the consensus timestamp of their genesis event, their operator index (e.g. 7th operator

[12]Subject to change

to join the network), and any other use-case specific metadata (e.g. number of failed audits in Shadow Drive).

3 Shadow Drive

3.1 Executive Summary

Remote storage has forever changed the way users interact with their data. It has enabled seamless curated user experiences across personal devices from anywhere in the world, and easy media and document sharing and collaboration among users. In addition, it has allowed users to recover data and device state in case of device loss or failure. In this sense, remote storage providers have become data banks; users entrust providers with their data, and trust that they will be able to retrieve and use it in the future – full stop. To this end, storage solutions must be robust, secure, resilient, accessible, and reliable to provide users with firm guarantees about data accessibility and integrity. As remote storage becomes increasingly vital for the majority of users and corporations, and as this digital infrastructure has grown into an integral part of our society and economy, these requirements have become critical and non-negotiable. An analysis and forecast by IDC [5] concluded that global data storage needs will exponentially grow to over 160 zettabytes of data by 2025 with a significant fraction of this data living on public clouds (e.g. Amazon Web Services (AWS) and Microsoft Azure). Not only is the amount of data stored by corporations increasing, but the proportion of this data which is sensitive (e.g. financial records, customer data, medical records) is increasing. In the 2023 edition of the annual Thales Group Data Threat Report [6], they report an increase from 49% in 2021 to 75% in 2023 of their respondents reporting that more than 40% of the data they store in the cloud is sensitive data. The modern storage systems that we are increasingly reliant on must respect these needs for privacy, durability, and availability.

General purpose distributed consensus systems like GenesysGo's DAGGER can orchestrate a storage system with this required safety and performance. DAGGER is designed without a single point of failure, allowing the Shadow Drive distributed system to reach Byzantine fault-tolerant consensus on all write operations. This decentralized design comprised of many nodes reduces the impact of individual node failures and downtime on Shadow Drive. Shadow Drive gives users sole and complete control over their data and privacy. Shadow Drive's client-side encryption model ensures user privacy, and guarantees that a data breach will never be the fault of the Shadow Drive network or system. Shadow Drive aims to be a storage solution that enables cold/archival storage and hot database-like use cases, achieved through careful design considerations impacting the system's storage efficiency and performance. It aims to be simple to use and easy to migrate to by offering an API that is compatible with Amazon Web Service's Open Source S3 API, and by offering an SDK exposed to many languages including Rust, Typescript, Python, and C. Additionally, it aims to offer a pay-once-and-store-forever immutable storage option, enabling low-cost permanent records and archives, and Web3 use cases such as digital assets pointing to media and documents. This flexible general purpose storage system can be used for a wide variety of applications including storing encrypted sensitive data, archival scientific datasets, training and validation sets for AI systems, historical blockchain transaction data, and much more.

The full Shadow Drive whitepaper to be released at a later date covers how DAGGER is used and adapted to enable Shadow Drive. In particular, it describes the Shadow Drive storage account model, the dual database on each operator node comprised of metadata and shards and their internal data structures, the hybrid erasure code and replication scheme used which enables high storage efficiency and data durability, and the data audit, repair, and rebalance processes carried out by the system which enhances data durability. It outlines the S3-compatible API and describes how it is implemented and translated into transactions which are to be consensus ordered by DAGGER. It also discusses how we establish a reputation system that aids in determining how ingested data will be distributed across the system, and how the economics align the interests of operators, auditors, and users to ensure long-term viability.

3.2 Mobile Network

When examining the history of the development of server-grade infrastructure, it's easy to see that remote storage was made possible by significant increases in bandwidth. Without it, extremely long download times would make remote storage unreasonable except for important documents and media. Mobile devices such as smartphones and tablets with cellular data have seen a similar trend. 5G, and eventually 6G, technologies that approach and exceed gigabit speeds coupled with modern fast multicore mobile processors and significant increases in mobile device storage capacities make it possible for mobile devices to be active participants in distributed systems. In fact, the specifications of today's top shelf smartphones resemble the specifications of personal computers from only 5-10 years ago.

Over 280 million smartphones were sold in the first quarter of 2023. Globally, there are over 7.5 billion smartphones with an active mobile network subscription. A typical modern smartphone has at least 64GB of on-device storage. The storage capacity of the iPhone 14 ranges from 128GB to 512GB, while the Solana Saga base specifications offers 512GB of storage. Capturing even just a few percent of this global storage capacity remains an untapped exascale opportunity. There exist challenges relating to battery life and uptime in incorporating these devices into a distributed system. By only relying on mobile devices for additional redundancy, mobile devices can easily be incorporated to strictly enhance Shadow Drive's durability guarantees without hindering the network's performance and data availability. To preserve battery life, it's important to realize that nodes that store data do not necessarily have to participate in consensus. This reduces the compute and energy requirements for mobile devices to be part of the network. Shadow Drive produces cryptographically secure hashes after DAGGER consensus replicated on all nodes for detecting data corruption and tampering. Since these guarantees apply regardless of where data is stored, this data retrieval from Shadow Drive remains verifiably secure regardless of whether retrieving data from an operator node or a mobile node.

4 Other Use Cases of DAGGER

DAGGER is a general purpose Byzantine fault tolerant consensus mechanism. As such, there are many use cases beyond Shadow Drive.

4.1 Smart Contract Platform

Similar to Bitcoin, DAGGER could be used to coordinate writes to a replicated ledger. If these transactions refer to executable code and data on the ledger, then a general purpose smart contract platform like Ethereum or Solana that is highly bandwidth-efficient can be constructed. DAGGER does not specify an execution architecture, although architectures like BPF or WASM are recommended due to their simplicity, portability/compatibility, and the developer tooling that already exists for these targets.

Building a blockchain with smart contract capabilities inherits all the safety guarantees from DAGGER, including being Byzantine fault tolerant. As applies to DAGGER, sybil attacks can be prevented with concepts borrowed from Proof-of-Stake; an event's vote which is cast on some other event can be weighted by the stake held by the event's creator. The stake could either have monetary value or be a scalar weight which is some function of the time since the operator's genesis event (or some other stake weight). Economic incentives must be aligned to dissuade operators from attempting to hold multiple identities. Penalties are another form of dissuasion.

4.2 Bridges and Oracles

State (or state changes) produced as a result of some consensus mechanism on some blockchain typically cannot be verified quickly or without relying on some trusted state on another blockchain. Naively, given two snapshots of a distributed ledger signed by a supermajority of operators prior to the desired state change, one could verify that a given state change was produced between the snapshots by replaying the consensus, ensuring the state change is present and that the second snapshot is obtained. Although this replay may be faster than the original consensus pass, it is still not possible to verify with low latency or within the typical computational constraints of a blockchain's transaction execution environment. Bridges aim to improve this process by reaching consensus on the observation of a state change on some blockchain, and reporting a proof of that state change on another blockchain. This proof could be permissioned such as a set of signatures from trusted and known parties as is done with the set of Guardians on Wormhole bridges. This proof could also be generated permissionlessly, and even opaquely but verifiably, as is done with zkBridge [4], where only a generated zero-knowledge proof must be verified.

An interesting observation to make is that when such a system carries out this process and propagates a confirmed message or event from one blockchain to another, it is called a bridge. However, when a system that does the same thing from a non-blockchain source to a blockchain, it is called an oracle. These are regarded as two different systems despite the fact that the underlying process is the same: a set of nodes collaborate to construct a proof of an event to be published on a blockchain. Being a general purpose consensus mechanism, DAGGER is capable of catering to either use case. Just as Shadow Drive required additional components and features like erasure coding and auditing due to its distributed storage design, these other use cases require additional components as well. A DAGGER-based oracle would still require specialized components for data aggregation and verification. Likewise, a DAGGER-based bridge, if based on light clients, would still require specialized components that make the bridge robust to 51% attacks, eclipse attacks, long range attacks, and other bridge vulnerabilities. In these cases, the primary function of DAGGER are:

- To allow for dynamic membership of the committee responsible for these functions, including to coordinate distributed key generation if required.
- To serve as the platform on which users submit requests (e.g. request to bridge).
- To keep a historical blockchain of previous observations.
- To submit signatures, transactions, or confirmations of some observation to the target platform.

5 Private Cloud Infrastructure

Byzantine fault tolerance is a property that is useful even when all nodes are trusted and run by a single entity. MUO defines BFT as "the ability of a network or system to continue functioning even when some components are faulty or have failed". Trusted nodes can still have failures due to corruption, external bad actors (e.g. man in the middle), power outages, hardware failure and so on. Private cloud infrastructure can be orchestrated using DAGGER. Deployments can look as simple as deploying a permissioned Shadow Drive on a set of trusted nodes, to more complex federated networks encompassing multiple organizations managing multiple compute and storage resources. In this case, DAGGER would keep keep track of queues and capacity of resources.

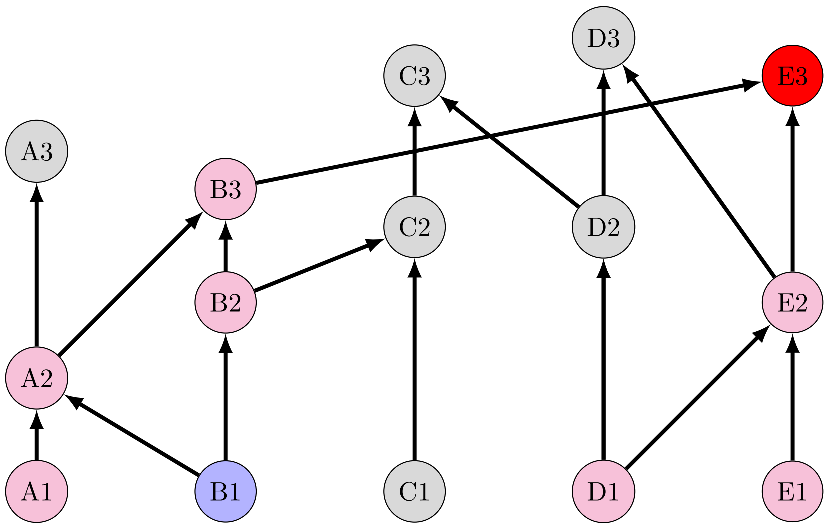

Figure 6: This shows a directed acyclic graph where all of the ancestors of E3 are in magenta, and those that are not ancestors are in gray. It is said E3 can see event B1, since it is an ancestor and there are no forks.

A To See and Strongly See

To reach consensus, the graph module must be able to evaluate whether an event E in the graph can **see** some other event X in the graph. To do this efficiently without having to inspect all events in the graph, we can perform a breadth-first search starting at event E. Pushing this initial event E to a max-heap with the event timestamp as the key, we iteratively take the top event and push an event's parent and self-parent to a max-heap (timestamp as key) until we encounter X (if we encounter X). Since the timestamp of every parent and self-parent must be smaller than the event's, we can terminate a branch in the search whenever the timestamp of a visited node is smaller than X's timestamp (i.e. do not append this parent or self-parent to the max-heap). The search terminates when we encounter X (in which case E sees X), or when the max heap is empty (in which case E does not see X). For an example, see Figure 6. The graph module must also be able to evaluate whether an event E in the graph can **strongly see** some other event X in the graph. With the **see** and ancestor iterator primitive outlined, the **strongly see** primitive can be implemented as follows. If an event E can strongly see X, then it can also see X (it is a weaker condition). As such, a similar breadth-first search is carried out. However, now, a buffer containing the latest event peer that E can see from each peer is collected along the way. If X is not reached, E does not strongly see X. If X is reached, and if exactly two-thirds or less operators are present in the buffer, E cannot strongly see X. If more than two-third of operators are present (call this threshold T) in the buffer of events, and more than T of these events can see X, then E strongly sees X. For an example, see Figure 7.

References

[1] Leemon Baird. Hashgraph consensus: fair, fast, byzantine fault tolerance. *Swirlds Tech Report, Tech. Rep.*, 2016.

[2] Pierre Chevalier, Bartlomiej Kaminski, Fraser Hutchison, Qi Ma, Spandan Sharma, Andreas

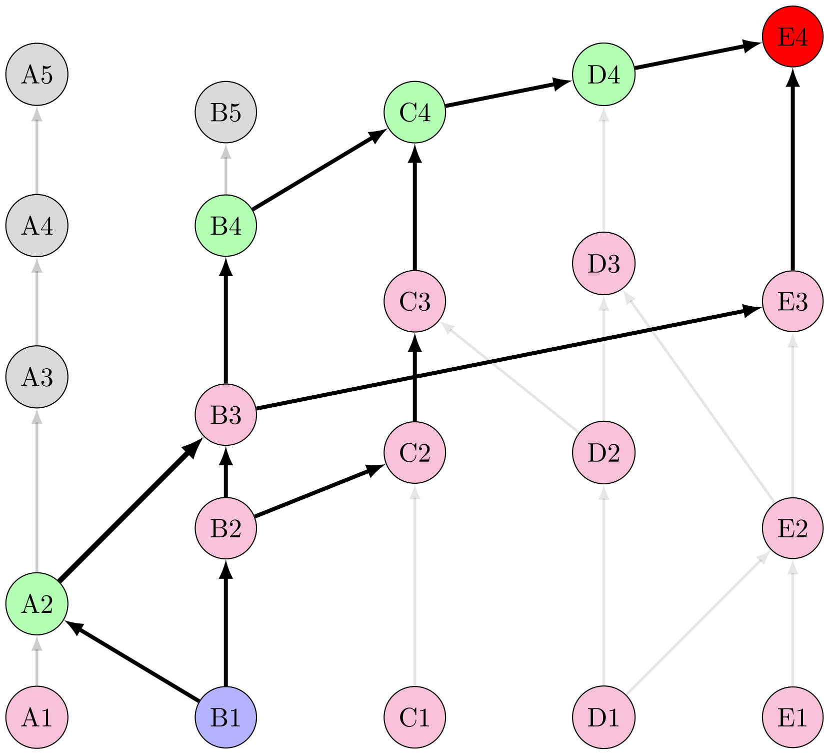

Figure 7: This shows a directed acyclic graph where all of the ancestors of E3 are in magenta, and those that are not ancestors are in gray. It is said E4 (red) can strongly see B1 (blue) if it has events created by more than two-thirds of peers which are ancestors that all see B1. E4 can see events A2, B4, C4, D4, and E4, which is more than two-thirds of peers, which all see B1 – these are the latest events by these peers which E4 can see, and are all in green. So, E4 can strongly see B1. Conceptually, this means that at the time of creation of E4, peer E knows that two-thirds of peers know about B1. Moreover, every peer that later adds E4 to their graph will know that peer E knows this.

Fackler, and William J Buchanan. Protocol for asynchronous, reliable, secure and efficient consensus (parsec) version 2.0. *arXiv preprint arXiv:1907.11445*, 2019.

[3] Future Market Insights Global and Consulting Pvt. Ltd. Cloud computing market size, share trends analysis report by service (saas, iaas), by end-use (bfsi, manufacturing), by deployment (private, public), by enterprise size (large, smes), and segment forecasts, 2023 - 2030, 2022.

[4] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical, 2022.

[5] IDC. Data age 2025: The evolution of data to life-critical, 2017.

[6] IDC. 2023 data thread report: Perspectives and pathways to digital sovereignty and transformation, 2017.