Tycho Litepaper

Broxus Team

v0.1.1

Content

Content **2**

Abbreviations and Definitions **3**

Introduction **4**

The Overview of DAG-based Protocols 4

IOTA, Avalanche, Hashgraph 4

PHANTOM, GHOSTDAG and BlockDAG 5

Mysticeti (Leaderless BFT with DAG-based Proposer Graph) 6

Execution-layer DAGs and Non-ledger DAG Usage 6

The Overview of Smart-Contract Virtual Machines 6

DAG Consensus & Collator **9**

Tycho’s 2-Phase-Commit 9

Mempool 9

Collator 12

Work Units 15

Virtual Machine & Smart Contracts **17**

Data Storage **20**

Serialization & Low-level Storage Optimization 20

State Management and Merkle Updates 21

State Node Adapter 30

BlockStrider 30

Scalability & Interoperability **33**

How Tycho Achieves Better Scalability 33

Queue State Separation from Shard State 33

High Performance Message Grouping 33

Minimized Data to Validate 34

Tycho-Based Protocols and TON 34

Network **35**

Benchmarking **36**

Comparison of DAG Protocols 37

Conclusion **39**

References 40

Abbreviations and Definitions

Key Abbreviations

2PC: 2-Phase-Commit.

aBFT: Asynchronous Byzantine Fault Tolerance.

ADNL: Abstract Data Network Layer.

AVM: Algorand Virtual Machine.

BFT: Byzantine Fault Tolerant.

DHT: Distributed Hash Table.

DLT: Distributed Ledger Technology.

EVM: Ethereum Virtual Machine.

FPC: Fast Probabilistic Consensus.

L2: Layer-2

LT: Logical Time.

DAG: Directed Acyclic Graph. A graph structure with directed edges and no cycles.

PoS: Proof of Stake.

PoW: Proof of Work.

TEAL: Transaction Execution Approval Language.

QUIC: Quick UDP Internet Connection.

RLDP: Reliable Large Data Protocol.

RPC: Remote Procedure Call.

SVM: Solana Virtual Machine.

TON: The Open Network.

TPS: Transactions Per Second.

TVM: TON Virtual Machine.

WU: Work Units.

VM: Virtual Machine. A computation engine that executes smart contract code in a sandboxed environment.

Key Definitions

Anchor: A special point in the DAG mempool that serves as a committed reference after three consecutive points form an "anchor pattern." Anchors belong to the global DAG and enable deterministic ordering of messages.

Bag of Cells: A collection (set) of cells.

Cell: The fundamental data structure in Tycho.

Continuation: an execution token in TVM that enables control flow graphs, exception handling, and the ability to stop and resume smart contracts.

External Message: Messages originating from outside the blockchain network (e.g., user transactions) that enter through the mempool.

Equivocation: The creation of alternative points at the same round by the same producer, which is detected and treated as a protocol violation. A “fork event” is a close equivalent definition.

Internal Message: Messages generated by smart contract execution that are processed within the blockchain network.

Mempool: The DAG-based first stage of Tycho's 2PC consensus that provides collating nodes with a common set of external messages, acting as a persistent ring buffer.

Point: The basic object of the mempool that carries external messages and metadata, essentially a vertex in a node's local DAG view.

Work Units (WU): A sophisticated measurement system that evaluates block release time by considering message processing, dictionary operations, VM gas consumption, account updates, and outgoing messages to synchronize collators with the mempool.

Introduction

Tycho is a novel blockchain protocol which extends core features of The Open Network (TON) by leveraging Directed Acyclic Graph (DAG) approach when receiving, processing and confirming messages inside TON. The concept of a DAG is a well-established construct in graph theory, defined as a directed graph with no cycles [1][2][3]. DAGs are widely applied in computer science domains such as scheduling, data flow analysis, and dependency resolution. In recent years, DAGs have also been adopted in the context of blockchain and distributed ledger technology (DLT).

Notable examples of the recent innovations in DAG-based designs include Bullshark[4], MYSTICETI[5], and Narwhal-Tusk[6]. They introduce efficient consensus mechanisms suitable for high-throughput and low-latency environments.

Sequential blockchains, such as Bitcoin and Ethereum, rely on a linear chain of blocks that imposes fundamental constraints on transaction throughput and confirmation latency. These limitations arise from the requirement that blocks be appended one at a time, with strict ordering enforced by consensus mechanisms like proof-of-work (PoW) or proof-of-stake (PoS) and its derivations. As transaction volumes grow, these linear architectures experience performance bottlenecks and increased confirmation delays.

Unlike global sequential chains, modern DAG-based protocols enable concurrent transaction validation and partial or asynchronous ordering, thereby enhancing throughput and reducing latency. Bullshark[4], MYSTICETI[5], and Narwhal-Tusk[6] etc. achieved high-performance consensus while maintaining robustness against faults and asynchrony. The adoption of DAG in smart contract platforms like Sui and Aptos underscores the shift toward scalable, production-grade distributed ledgers. With Tycho, TON protocol adopts DAG for the purpose of its own original smart-contract virtual machine.

However, the design and analysis of DAG-based systems also introduce new challenges. These include transaction ordering ambiguity, consensus safety under concurrency, finality guarantees, and complex network behavior under adversarial conditions. Additionally, the lack of standardized evaluation frameworks complicates the comparison of protocol properties across different implementations.

The implemented Tycho DAG mainly follows the principle “the code is specification” and is freely available on GitHub [8] for more elaborate analysis.

The Overview of DAG-based Protocols

In this section, we describe the architectural designs of the most prominent DAG-based systems, focusing on their graph structure, consensus mechanism, and execution model.

IOTA, Avalanche, Hashgraph

IOTA is a DAG-based distributed ledger that replaces blocks with a transaction graph called the Tangle, where each transaction approves two others[9], enabling feeless, lightweight microtransactions. Originally dependent on a centralized Coordinator to prevent double-spends, IOTA's Coordicide redesign introduced a leaderless consensus via Fast Probabilistic Consensus (FPC) and a reputation metric called Mana, which replaced fees but relies on strong assumptions about Sybil resistance and user behavior[10]. While the Tangle supports parallelism, its lack of total ordering complicates smart contracts and yields only probabilistic finality, and FPC's robustness under network delays remains uncertain. Despite ecosystem growth[11], IOTA has faced criticism over security, liveness, and resilience in adversarial settings.

Avalanche introduces a family of consensus protocols based on repeated subsampled voting over a DAG of transactions[12], where nodes query random peers to determine preferences and gradually converge toward consensus. This approach enables high throughput and fast confirmations under favorable conditions, but its probabilistic safety means consensus is not guaranteed under asynchrony or coordinated attacks[13]. The protocol requires careful parameter tuning to balance liveness and safety, and while Avalanche's multi-chain architecture - with the X-Chain for assets, C-Chain for smart contracts (using the linearized Snowman protocol), and P-Chain for validator coordination - offers flexibility, it also increases complexity and raises questions about cross-chain interoperability and consistency[14].

Hashgraph is a DAG-based consensus algorithm that uses gossip and virtual voting to achieve asynchronous Byzantine fault tolerance (aBFT) without blocks or mining[15]. Nodes create events referencing two parents - one local and one received - forming a DAG that preserves causality, and consensus is reached via deterministic virtual voting without actual message exchange[16]. While the protocol guarantees finality under full asynchrony with less than one-third malicious nodes, it relies on a permissioned governance model through a fixed set of council members, limiting decentralization. Additionally, it assumes all nodes have full visibility of the DAG, which may not hold under high churn or adversarial conditions, though it offers a theoretically robust consensus layer with deterministic ordering and low latency.

PHANTOM, GHOSTDAG and BlockDAG

PHANTOM is a DAG-based consensus protocol that extends Nakamoto's longest-chain rule by defining a total order over a blockDAG using k-cluster (kk). It defines a set of blocks connected to all but at most k others which are topologically sorted to form the ledger[17].

While finding a k-cluster is NP-hard problem, PHANTOM employs the GHOSTDAG heuristic, which selects a "blue" set of blocks based on connectivity to represent the honest chain, enabling higher throughput through parallelism. However, PHANTOM and GHOSTDAG are vulnerable to incentive attacks if miners deviate from prescribed transaction selection, risking reward manipulation and reduced decentralization for the whole network[18]. Building on GHOSTDAG, the BlockDAG platform integrates UTXO and account-based models, allowing smart contracts and traditional transactions to coexist with a bridging mechanism between them[19]. While this hybrid architecture improves flexibility and throughput, it introduces complexity in maintaining consistency and requires further analysis to ensure security, double-spend resistance, and performance under adversarial conditions.

Mysticeti - Leaderless BFT with a DAG

Mysticeti is a DAG-based consensus protocol that achieves deterministic finality in partially synchronous networks without a fixed leader[5]. It allows all nodes to propose blocks concurrently by organizing proposals into a DAG, where each proposal links to earlier ones, forming a proposer graph. Consensus is reached by identifying "committable" proposals through quorum intersection and acknowledgment accumulation, enabling agreement without leader rotation or slashing. The protocol guarantees safety and liveness under the stake-weighted BFT assumption and does not rely on randomness or probabilistic finality. While Mysticeti avoids equivocation and improves parallelism, it assumes a fixed validator set, and its applicability to dynamic or permissionless environments remains an open question, along with the added complexity of tracking proposals and evaluating commit rules.

Execution-layer DAGs and Non-ledger DAG Usage

Certain blockchain platforms, including Sui and Aptos, make internal use of DAG structures to improve execution-layer parallelism. These systems employ consensus protocols such as Bullshark[4], which leverage DAGs to decouple data dissemination from consensus, but do not expose a DAG-based ledger structure to end-users. Instead, DAGs serve as auxiliary tools for transaction scheduling and execution ordering within otherwise traditional BFT frameworks[20][21].

In both Sui and Aptos, transaction dependencies are modeled as causal DAGs to enable speculative parallel execution. However, global ordering and state finality are achieved via consensus mechanisms that produce linear ordered chains.

The systems reviewed in this section demonstrate the diversity of architectural approaches to DAG-based consensus. While some protocols adopt DAGs as the core structure for global ordering and transaction validation, others utilize them locally to optimize execution-layer concurrency. These differences highlight a broad design space in which trade-offs arise between performance, consistency, security, and implementation complexity.

The Overview of Smart-Contract Virtual Machines

Tycho’s Virtual Machine (VM) implemented ideas initially proposed for TON VM and therefore significantly different to any existing protocol. The VM's advantages stem from both its underlying TVM architecture and design choices, and Tycho's specific optimizations for parallel blockchain execution which is different to any known to date smart-contract platform. In the present section we give a brief overview of the most popular VMs.

The Ethereum Virtual Machine, initially proposed in 2015 to overcome the limitations of the Bitcoin Script, has been a long-standing leader in setting common standards for smart contract developers. It has successfully abstracted away the complexity of the computation engine from the process of building sophisticated smart contract systems that compile to Ethereum's instruction set. The share of developers working in Ethereum ecosystem (#1 across the globe [22]) exposes it as the clear winner in terms of adoption of a programmable blockchain platform if compared to its predecessor Bitcoin.

As the first mover in terms of shaping the “Smart contract Virtual Machine” concept, the EVM must have addressed some fundamental problems. Due to the halting problem, the EVM needed variable gas costs to meter the execution of smart contracts with cycles as they go. Other key characteristics of EVM include: a 1024-depth stack for operands, 256-bit word size, and the stack architecture which means instructions push/pop values on a LIFO stack for operations(e.g. an ADD pops two numbers and pushes the sum). EVM is single-threaded globally and consequently has performance limitations which may be addressed via adding different computational layers on top of the base layer. Despite that, it is now the de facto standard for many Layer-1 and Layer-2 (L2) chains.

Figure 1. Types and flavors of purpose-built virtual machines for smart contracts and blockchain protocols.

Such lesser-known projects as Tezos – Michelson (Stack VM for Formal Verification) and NeoVM seem to be underrepresented in the literature. Whereas having their own distinctive features, both VMs make an attempt to guarantee EVM compatibility as reported by the documentation [23][24]. Similarly to Bitcoin Script, Michelson operates as a stack machine. it has no variables, only a stack that instructions manipulate. The Tezos VM interprets Michelson opcodes in sequence, modifying the stack. The

execution model of NeoVM is explicitly stack-based, too. However, it has multiple stacks, which makes it fairly different to other solutions. The NeoVM operates with an Evaluation Stack for operands and results; an Invocation Stack for managing call frames (each contract call or function call pushes a new execution context on the invocation stack); and a Result Stack for returning results. Neo is not reflected by the Electric Capital Developer Report 2024 [22] which indicates its low adoption or potentially very closed community.

Algorand #32 follows Tezos #30 in the 2024 developer rank [22] but its consensus and virtual machine design attracted the researcher's continuous attention since 2016 [25] when it was initially proposed. The Algorand Virtual Machine (AVM) runs programs written in TEAL (Transaction Execution Approval Language). Bartoletti et all [26] demonstrated an application of developed declarative smart contract language for formal verification of fundamental properties of the underlying blockchain protocol. A higher-level PyTeal library for constructing TEAL scripts is also mentioned by the authors. The AVM runs these instructions sequentially in a stack. It closely resembles a simplified Forth-like VM. TEAL opcodes cover arithmetic, logic, crypto (hash, signature verification), flow control, and state access (reading/writing local or global state). Algorand continues to evolve the AVM (currently documentation suggests v9 version as the most recent).

The Move VM was developed for Facebook's Libra and evolved into Aptos and Sui networks. The distinctive feature of the Move Smart Contract language became a top-down design with a strong focus on security and safety. A group of researchers introduced a new smart contract VM tailored to a resource-oriented programming model [27] and later proposed tooling for easy proving of developed smart contracts which may be facilitated along with testing them [28]. The Move VM guarantees that 1 coin in a resource variable will never be duplicated or lost. By analyzing which resources a transaction touches, the executor can run multiple Move transactions in parallel.

The Solana VM (SVM) is the first VM that attempted to widely adopt a host’s runtime environment for parallel execution of smart contracts in a blockchain protocol aimed to achieve maximally possible throughput and lowest latency. Solana uses a unique execution environment designed for massive parallelism. "Programs" on Solana are written in Rust (or C/C++), compiled to Berkeley Packet Filter (BPF) bytecode, and executed by Solana's BPF interpreter or JIT on-chain. A similar approach but applied to IoT [29] demonstrated the substantial benefits of register-based VM over WASM's stack-based design. Inside Solana's VM, thousands of contracts (BPF programs) could run in parallel across the validator's CPU cores with little overhead to the validator node. According to Zandberg and Baccelli [29], the BPF VM design required only 10%

more memory for running an application in the VM than running it directly on the microcontroller without VM.

DAG Consensus & Collator

Initial design of the TON consensus considered non-guaranteed deliverability and processing of *messages* by nodes along with block production. Tycho preserves non-guaranteed deliverability as a design principle for increased throughput while addressing internal availability via introduction of 2-phase-commit (2PC) approach to message processing and transaction settlement. The important place in the architecture belongs to DAG topological structure remotely resembling the Unspent Transaction Output design approach.

Tycho's 2-Phase-Commit

The introduction of Tycho DAG and its implementation in Rust followed a re-implementation of Catchain (short for Catch-Chain [30]) in Rust. This is a Byzantine Fault Tolerant (BFT) protocol designed for asynchronous block production and finalization in a decentralized validator set in the presence of network delays or malicious nodes. It was originally proposed for The Telegram Open Network (currently The Open Network [31]) by N. Durov [31]. It ensures that all honest validators agree on the same block history, aiming to get a stake-weighted majority of signatures for a block through stages of proposing and voting. Although the whitepaper [31] mentions potential utilization of DAG topology with regard to Shard state or generally any abstract cell tree representations, early implementation did not introduce it. Therefore no deduplication of external messages before execution took place and the consensus layer relied solely on replay protection in smart contracts; also no guarantees were made on external messages delivery. Thus, Tycho may be seen as an implementation of very early ideas envisioned for TON, with more specifics applied due to acquired experience.

In the proposed two-stage approach, a pending external message first enters the Mempool, which is structured as a DAG, and may be processed by the Collator. Previous works, Google's Spanner [32], Narwhal/Tusk [6], and the openCBDC project [33] highlighted the benefits of two-phase commit (2PC), despite differences in the implementation details of each protocol. It is important that in the openCBDC (Project Hamilton) initiative, the authors implemented and evaluated two alternative architectures: the atomizer architecture and the 2PC architecture. They concluded that while the atomizer provides a globally ordered history of transactions, it suffers from limited throughput. The 2PC architecture, by contrast, achieved near-linear scalability

in peak throughput with additional resources, but it did not preserve a globally ordered list of transactions. Similarly to openCBDC, the Tycho DAG adopts a localized view of transaction history, enabling parallel execution of non-conflicting chains of messages that trigger smart contract execution.

Mempool

The purpose of the DAG-based “Mempool” in Tycho is to provide collating nodes with a common set of external messages. In very general terms, it acts as a persistent ring buffer with a short lifespan of the containing data, as opposed to collated blocks, and serves as a distributed merger of iterators over external message queues kept in every collating node until they become a part of DAG. This section provides some details about DAG inner workings.

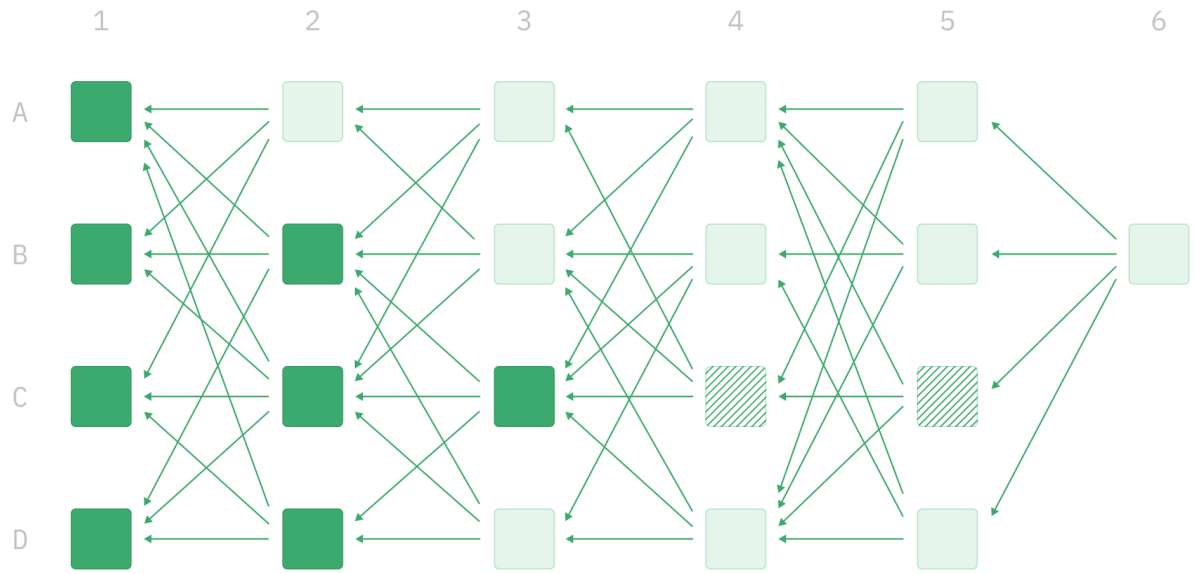

Figure 2. local DAG of point references viewed by node B at round 6; global sub-DAG is determined by points C3-C5 where C3 is an anchor; B6 is a source vertex and round 1 contains sink vertices pic source.

We begin with defining a "point" as the basic object of the mempool that carries external messages and metadata. It is essentially a vertex in a node's *local* DAG, but it is not necessarily included in the *global* DAG yet which is a result of their intersections. Every node builds its local view of DAG according to the references to the previous round specified in each received point. The requirement for nodes to agree if some point becomes a part of the global DAG demands a consensus-like algorithm.

Each node produces a new point during every broadcast round, provided it hasn't fallen behind the majority of nodes. After broadcasting each point, a node requests signatures from other nodes. Each signature serves as a guarantee of the point's

validity, persistence, availability, and referenceability. Nodes will only sign points from their current round or the immediately previous round. When the majority completes one round and begins the next, lagging nodes abandon the previous round and advance directly to the current one, since there's no benefit to producing points for outdated rounds.

When a node has signed a point from the previous round, it must explicitly reference that point in any new point it creates for the current or the next round. Additionally, every point must reference the majority of points from the previous round. For points from two rounds prior, a node references only those it has previously signed but not yet directly referenced.

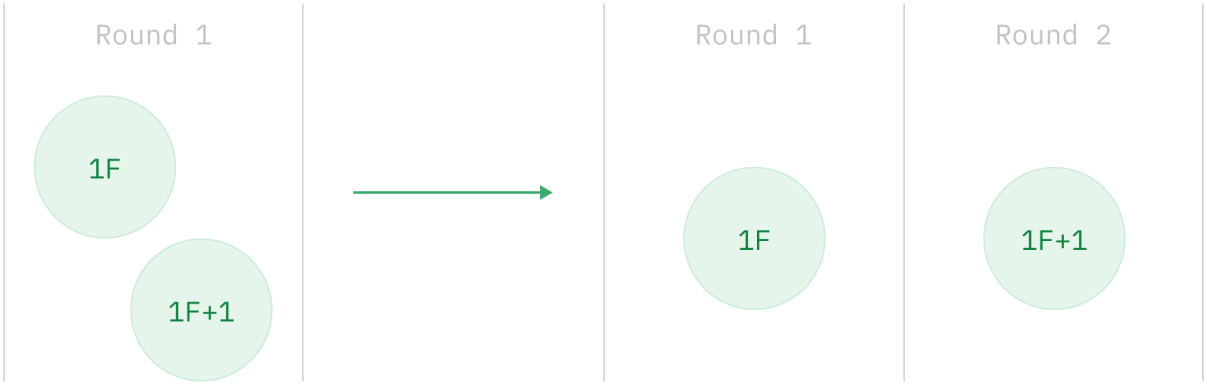

Figure 3. If nodes sign only the current broadcasting round, then in the absence of the unreliable minority consensus may stall, because some nodes may have accumulated enough signatures and references to advance their round while the others haven't. pic source.

Gaps may appear in the sequence of points when a non-malicious node joins mid-session. In such cases, the absence of signatures is permissible. However, when a node creates points in two consecutive rounds, the later point must include signatures for the previous one, or it will be invalid. Consequently, nodes must skip at least one round when they cannot include the required signatures.

Each skipped round represents a failure to fulfill the requirement of referencing a signed point. Other nodes will detect this behavior through DAG path analysis (along with ignored signature requests), increasing the lagging node's likelihood of being classified as unreliable or malicious.

A majority of signatures also ensures the uniqueness of signed points: within a single slot (defined by producer node and round), only one point can obtain it. However, point equivocation, i.e. when alternative points coexist in a single slot, can be referenced by different points from distinct slots, each of which may acquire majority

signatures. The protocol will detect and treat equivocation patterns as violations of consensus and corresponding penalty will follow.

The consensus mechanism considers only quorum nodes and is not stake-weighted. A full quorum consists of 3F+1 nodes, where at most F nodes may be Byzantine (exhibiting arbitrary behavior), while 2F+1 nodes are expected to be reliable. This approach guarantees that within a few rounds, any point will have a path to the first of two consecutive points through quorum intersection. On the Figure 4 if a point at round 1 is signed by 2F+1 nodes, then at least 1F+1 points (because 1F among 2F+1 may be byzantine) will reference it directly at round 3 (a point may be signed as from previous round only), so any point at round 4 will have a path to it (due to mandatory 2F+1 references from previous round). If a point at round 2 exists to carry those 2F+1 signatures, it is a marker of the described pattern.

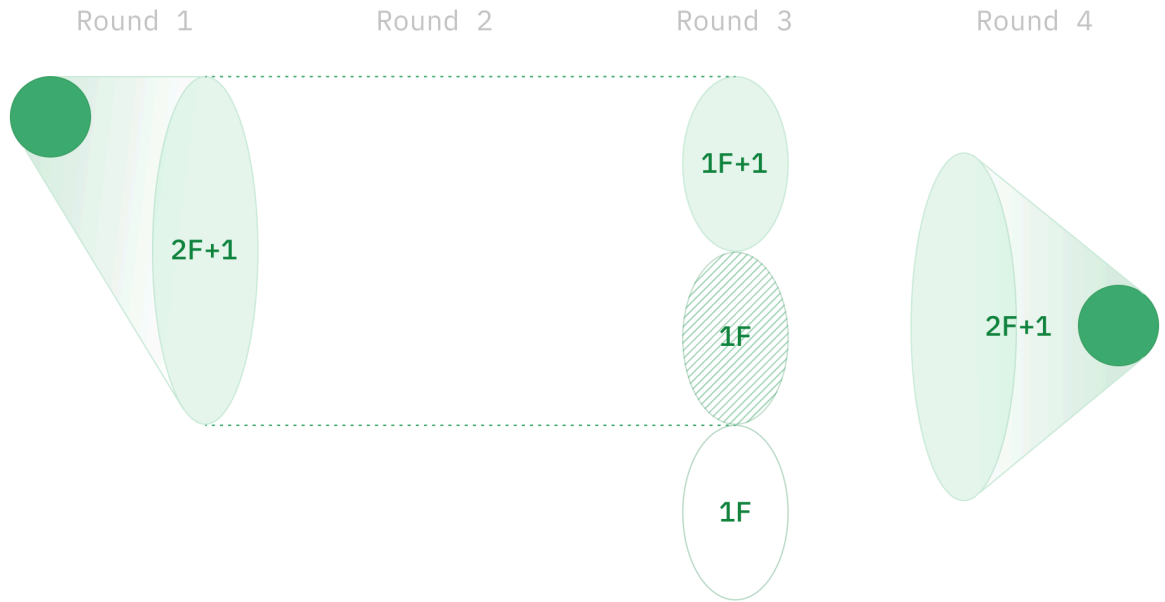

Figure 4. Quorum intersection. pic source.

Since any point may represent an equivocation, three consecutive points provide a guarantee: the first point is uniquely referenced through the second, and the second will be referenced by any subsequent point. These three consecutive points form the so-called *"anchor pattern,"* where the first point serves as an anchor that belongs to the global DAG along with all its references. The guaranteed path existence enables anchors from every fourth round to form a reproducible chain at any node, allowing lagging nodes to synchronize.

Every four rounds, a random node is selected as the leader. When that node produces three consecutive points, the resulting anchor is committed to the DAG. This means at least a majority of nodes observe a common sub-DAG and traverse it identically to

establish a consistent ordering of referenced points while preserving the historical sequence of rounds. A commit represents an irreversible state change that defines the data passed from the mempool to the collator, or change from the 1st phase to the 2nd.

External messages are extracted from each ordered point and must undergo deduplication which is done using a sliding window of rounds in a dedicated structure. Duplicates may happen because validators notify their closest neighbors about new external messages before they reach the mempool, resulting in redundant data that occupies part of the mempool's throughput to ensure reliable delivery.

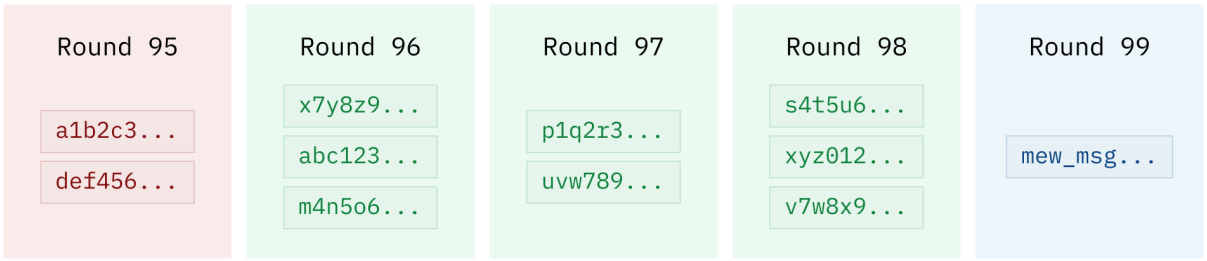

Figure 5. Illustration of sliding window of rounds carried out by Deduplicator.

However, the Collator receives deduplicated and ordered external messages uniquely identified by the anchor round. Setting aside the Collator's internal workings for now, it ultimately produces identical blocks on every node because collation is deterministic and the input data remains consistent across at least a majority of nodes. Each signed master block contains the same last processed anchor ID, which is reported back to the mempool. In turn, the mempool cannot initiate rounds significantly beyond this anchor value. This feedback mechanism ensures the mempool broadcasts at rounds below a certain threshold, enabling seamless validator set changes that are planned for and executed at that designated round.

The DAG consensus configuration changes are possible but it requires a mempool restart (without restarting the entire node) because point validation rules change in incompatible ways. The new DAG is initialized with a newly generated common genesis point, and the state machines resume operation from scratch.

Collator

The previously described DAG-based “mempool” manages the first stage of the two stage message commitment. In the second stage Collator creates new blocks and simultaneously conducts higher level computations (see Figure 6). While the DAG layer

does not execute transactions directly, the Collator is a system component that collects external and internal messages, processes them according to their logical timestamps, and forms execution batches (blocks). It acts as the orchestrator, preparing input data for execution of smart contracts in the Executor and ensuring that messages are processed deterministically. The collator also assembles blocks and notifies the DAG layer in a feedback loop with the last processed anchors, blockchain config changes, etc.

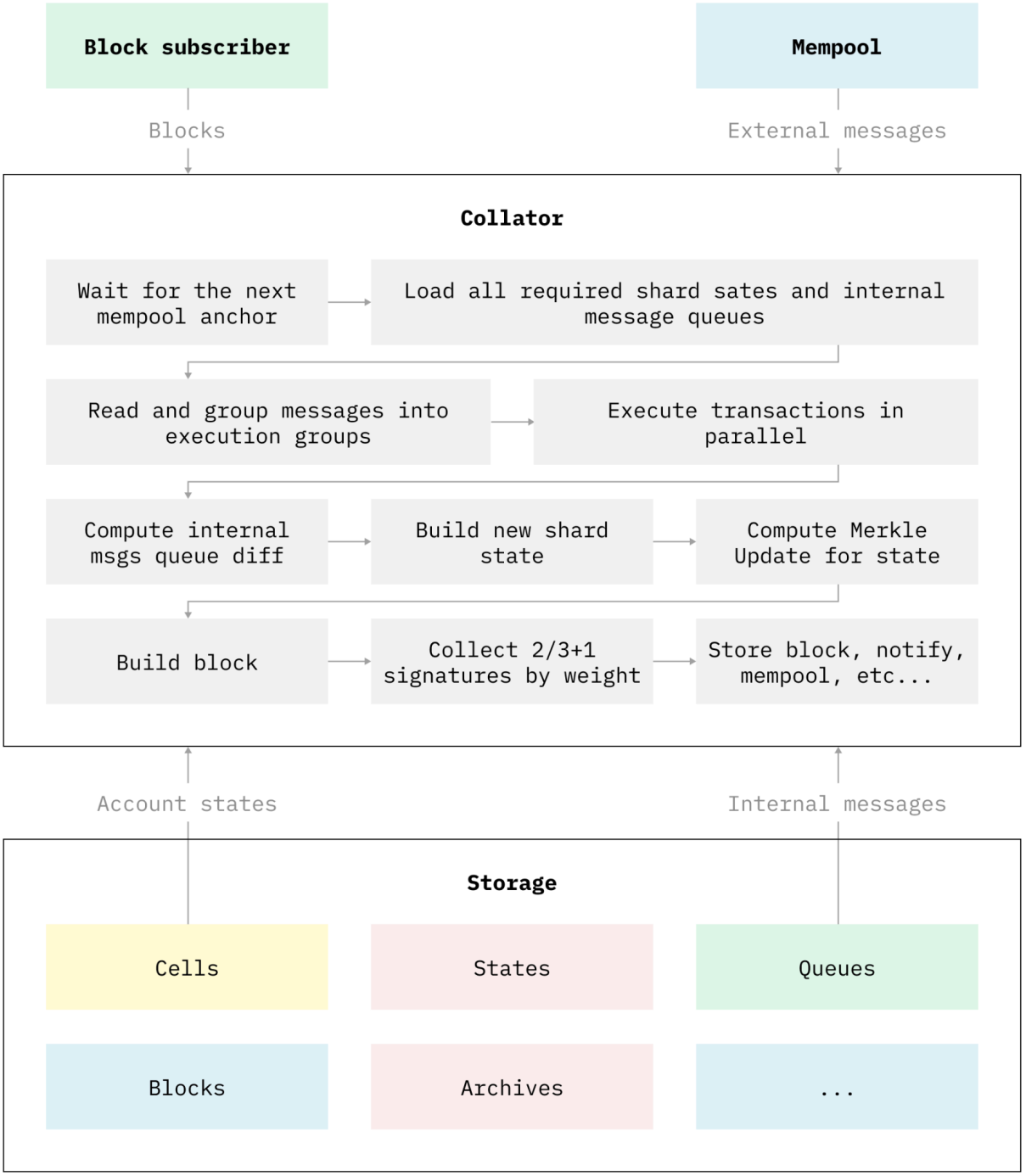

Figure 6. The Tycho Collator and its workflow.

The diagram on Figure 6 demonstrates the main stages of the block collation process which results in a new block. At the beginning collator reads available internal messages and receives new external messages from the Mempool. It groups messages for execution in such a way that maximizes the number of concurrently executed transactions, the Virtual Machine instances execute these messages on multiple accounts, the Collator saves transactions, changes account states and collects new internal messages produced as a result of executed smart contracts. While computational and size limits of the block are not reached, the Collator repeats the cycle of reading, grouping and executing messages.

Block limits include two main variables which indirectly limit its size in bytes/bits:

- Gas usage (for transaction execution);
- Logical time (LT) delta.

The Collator terminates the current block construction process when one of these thresholds is reached. Limiting block size is not considered as a priority in Tycho because this constraint may implicitly follow other constraints such as amount of accounts to be processed or number of transactions or external messages to collate.

The current Collator configuration implementation adopts the same model as TON, maintaining so-called soft and hard limit definitions in the config. However, the current system virtually disables hard limits, eliminating the rigid internal-then-external message prioritization. This streamlined approach allows continuous block collation until any hard limit is reached, removing the need for soft/hard limit distinctions in practical operation.

On the final stage the Collator forms an updated state, computes a Merkle update and finalizes a new block candidate, which combines various data and pointers:

- The block data (transactions, state updates, etc.);
- Reference to the masterchain seqno;
- Indication of whether it's a key block;
- Previous block IDs;
- Top shard block IDs;
- Value flow information;
- Queue diff information.

Since the first level commitment (via consensus) at the mempool level guarantees the order of external messages that the collator receives, and message grouping for execution is performed strictly deterministically, all nodes participating in consensus must produce absolutely identical blocks.

This block candidate can then be validated and, if valid, added to the blockchain. During the validation process network nodes exchange signatures of the issued block among themselves and once ⅔+1 signatures are accumulated by the weight of the stack, the Collator considers the block valid. If a node fails validation, then it begins synchronization and the collator will receive a valid block issued by the majority of other nodes. It will not be the same as the node's own block to be released. The collator cancels the current active collation of the next block, deletes the incorrect block and all uncommitted changes, loads the correct state from the database and continues working from this state.

For validation purposes, nodes exchange minimal data between themselves: a node's public key and a signature of the block's root cell hash. Each node uses the hash of its own next produced block to verify another's signature. This way nodes do not broadcast the actual blocks over the network. The exchange of signatures is performed very quickly over the network. However, if a node has not performed block collation, then it doesn't have the source data to verify the signature. It forces network participants to join the second stage commitment and include messages into blocks deterministically.

Deterministic execution of messages during collation requires certain rules and constraints which provide deterministic ordering. The execution ambiguity must be prevented, and the consistent state transitions across all validator nodes must be achieved.

Therefore, messages within a contract must execute in strict logical time hash order, ensured through outbound message logical time rules and message selection rules.

Outbound message LT rules consist of:

- Rule $LT(M_1) > LT(M_o)$, which means that sequential messages M_1 and M_o from contract A maintain monotonic logical time ordering.
- Rule of block-level ordering, which means that messages in block B+1 have higher LT than in block B+0.

Additional message selection rules include:

- Sorting by ascending logical time hash per recipient account for queued messages in the buffer.
- Strict logical time hash ordering for execution groups.
- Ordered execution in cohorts and guaranteed temporal precedence in separate buffers.

Resolution of co-dependent states is especially important. For them a “Triangle Constraint” ensures a specific order of interdependent state transitions. Let’s consider a situation when contract A simultaneously sends message M1 to B and message M2 to C, and B's execution of M1 generates another (internal) message M3 to C. Then according to this constraint, C must process message M2 before message M3 (Figure 7).

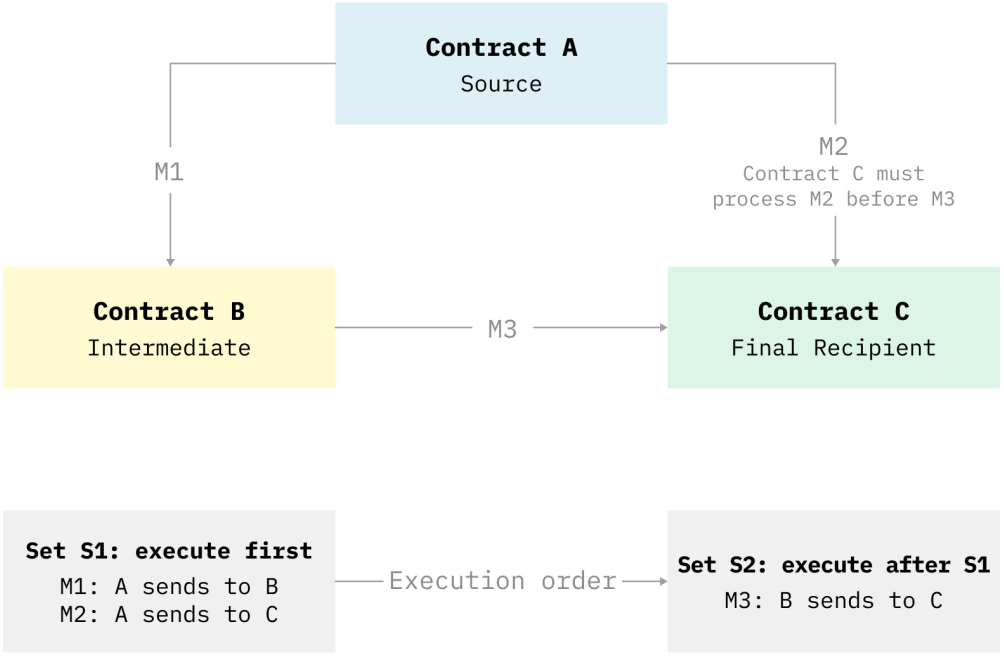

Figure 7. An Illustration of “Triangle constraint” rune for interdependent contracts with simultaneous calls.

The constraint is implemented as follows: M1 and M2 enter a set S1. Message M3 enters the subsequent set, ensuring M2 executes first regardless of whether processing occurs within the same block or across blocks. Cross-block scenarios maintain ordering through queue precedence over new message sets.

On the final note, Collator also implements a set of technical decisions which help to improve its performance and protect from denial of service type failures while working together with the DAG layer. These include:

- Isolation of large internal messages queues into dedicated partitions;
- Separating queue state from shard state using minimalist structures for state storage and diff exchange;
- Maximizing parallel transaction execution in the VM through efficient message buffering and grouping;
- Minimizing bandwidth requirements for peer-to-peer connections;
- Implementing parallel Merkle update computation and shard state change persistence (see Data Storage section).

The low level technical implementation of the Collator follows the principle of maximally possible parallelization of operations. This principle adds up to outlined architectural decisions specifically relevant to blockchain protocol that aims to be highly performant.

Work Units

To produce identical blocks on the same block height synchronously, all nodes should have ordered external messages incoming from the DAG mempool. Their collators must have an identical set of both anchors and (external) messages. *Work units* provide a feedback mechanism for collators to keep up with network conditions. However, let’s consider first the challenges arising from the requirement for all nodes of the network to have identical sets of anchors and external messages in more detail.

The timing of anchor and block processing creates significant synchronization challenges in distributed blockchain systems. Since different nodes experience varying delays when receiving anchors and releasing blocks, it's impossible to import anchors based on system time without risking nodes desynchronization in the network. While releasing an anchor in the mempool typically takes 1000-1200 ms compared to the much faster 400 ms block release time, importing anchors before every block would result in unnecessary 600 ms delays. However, implementing a fixed interval approach (such as importing anchors every N blocks) proves equally problematic - for instance, importing every 3 blocks at 500 ms intervals (1500 ms total) would lag behind the 1200 ms anchor release cycle, causing the collator to fall behind the mempool. This lag increases the time difference between when externals enter the mempool and when they're processed, degrading network latency for users and causing DAG size growth. The situation becomes critical when the mempool pauses after exceeding the maximum allowable lag threshold relative to processed anchors: if the last processed anchor (A) is A+1 while the last imported is A+218, and

the mempool pauses at round R+221 (with a 220-lag limit), the collator's request for the next anchor after A+218 will cause a complete network deadlock since the paused mempool cannot provide anchor A+222.

To synchronize with the mempool, the collator evaluates the time spent on block release in units called *Work Units (WU)*. The WU calculation formula takes into account the number of read messages, the number of dictionary operations, gas consumed in the VM, the number of updated accounts, outgoing messages, etc. The coefficients used in the formula are specified in the network configuration. This way, each node evaluates block release time in WU identically, regardless of the actual time spent. The network configuration also sets an estimated average time for releasing one anchor measured in WU. As a result, if there is increasing load and the accumulated amount of WU's during the release of previous blocks exceeds what's necessary for releasing one anchor, the collator imports an anchor before the next collation. If enough WU's have accumulated for N anchors, then N anchors are imported at once. This way, all nodes will deterministically import the next anchor after the same block at height seqno.

Currently, the WU calculation coefficients are set manually based on conducting numerous network tests under various loads. The formula isn't perfect, so estimation accuracy may change if the load profile changes. If the estimation is inaccurate, the collator may still "idle" or lag behind. Therefore, it's necessary to implement functionality for adjusting WU calculation parameters based on consensus between nodes.

Each node, when importing an anchor, determines whether the collator lags behind the mempool by checking if the anchor waiting time falls within a specified range. If a node regularly makes errors in estimating block release time in WU, it "votes" for parameter adjustment. Votes are sent as special external messages on behalf of nodes to a special contract that, upon reaching quorum, performs WU parameter adjustment in the network configuration. This way, automatic adjustment of WU calculation parameters will occur to keep the collator synchronized with the mempool.

However, the problem of possible node hanging when the collator requests the next anchor remains unaddressed. To resolve the issue, the collator obtains the current last processed anchor from the previous master block. Then it compares the last processed anchor with the last imported one. If the last imported anchor exceeds the last processed one by ⅔ of the maximum allowable lag (~146 rounds), the collator stops importing anchors to clear accumulated externals and move the boundary of the last processed anchor. This way, the last imported anchor will never be higher than ⅔ of the allowable lag, meaning the collator can't be hanging on anchor import permanently.

While traditional blockchains like Ethereum use gas limits primarily to constrain transaction execution within blocks, Tycho's Work Unit system serves as a multi-dimensional synchronization mechanism that accounts for message processing, dictionary operations, VM gas consumption, account updates, and outgoing messages to deterministically estimate block release time across all nodes. This approach transforms the static gas limit into a dynamic feedback system that enables precise mempool synchronization, ensuring all collators maintain identical anchor import timing regardless of hardware variations. By incorporating automatic parameter adjustment through consensus-based voting, Work Units effectively solve the distributed timing challenges that simple gas limits cannot address, making them a more advanced and adaptive version of the fundamental resource constraint mechanisms that govern blockchain block production.

Virtual Machine & Smart Contracts

The Tycho's Virtual Machine follows major principles outlined across the whole TON whitepaper document. A potential better outcome for interoperability between different networks that have adopted flavors of TVM is an important feature to consider here. The original principles turned out to be general enough to allow customization of popular Solidity smart-contract language to TVM and its step-by-step improvement. A reference implementation of Tycho VM and so-called Executor are open source and available on GitHub [34][35]. The original TON VM whitepaper does not mention more practical aspects of the TVM which are rather important for application developers and may not exist in other VMs. The present section of Litepaper attempts to fill this gap and sheds light on how smart-contracts work in Tycho at the node's level.

The protocol diagram on Figure 8 depicts the relation between Collator and Executor during execution of the message group, and suggests it as part of the Collator. The Executor is a pure function-like component that applies transactions to the system state. Given the input parameters (for example, account states and external messages), it produces a new account state and transaction results. In the present document, the Collator section mentions the Executor as part of general message processing that consists of different phases.

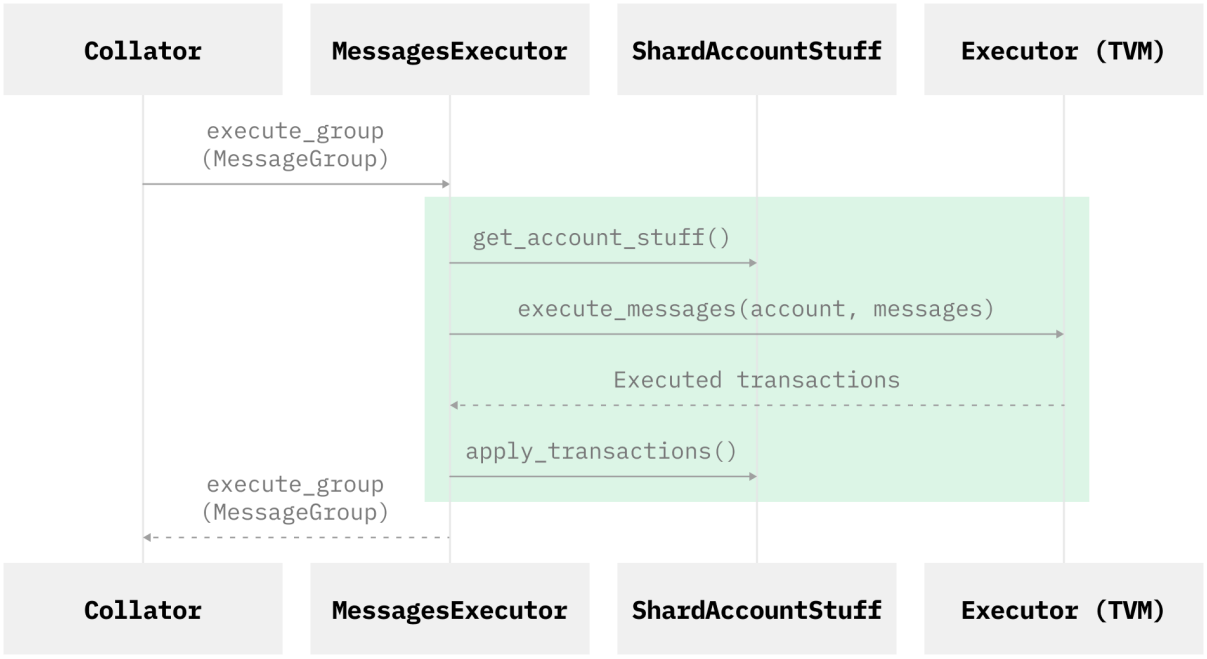

Figure 8. Message group execution workflow during block collation.

It handles various types of transactions. These are:

- normal account messages with data and smart-contracts,
- special transactions which create new assets and recover transactions,
- tick/tock transactions that execute system transactions for fundamental addresses,

It also converts results into new messages for the output queue of the smart contract.

The Executor initializes during the so-called Prepare phase with blockchain configuration and executor parameters. This way it enforces rules like gas limits and validity checks. During the Execution phase, it processes messages in sequence: “tick-” transactions, special transactions, then regular message groups, and then “-tock” transactions. Importantly, the executor launches the virtual machine when needed and parses its results.

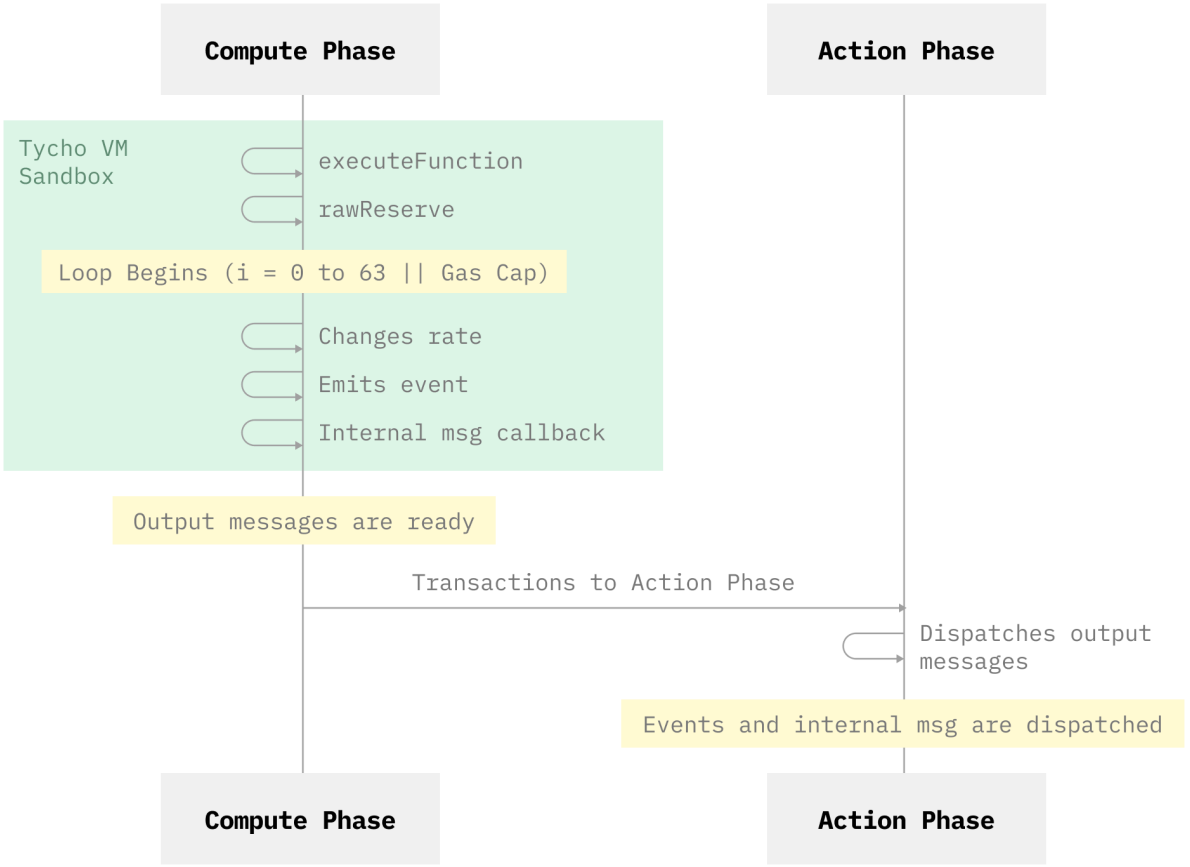

Figure 9. Tycho VM Sandbox and main message’s lifecycle phases: Compute and Action.

On Figure 9, the Executor wraps the Tycho Virtual Machine which reflects an actual architectural decision: hundreds and thousands of TVM instances, or “sandboxes”, could run in parallel and carry smart contracts through different phases, most

importantly Compute and Action phase. Besides, an ordinary transaction flow includes the Credit phase, Storage phase, and Bounce Phase (mentioned in the opcodes section of the whitepaper [31]) but only Compute and Action phases are practically relevant to Tycho VM. On the scheme, they are shown as the most crucial during the execution of the smart contract. The ”Executor” manages all other phases, too.

The VM operates strictly on the stack and registers as inputs and produces an updated stack, exit codes, and possible register modifications for the Action phase (see Figure 9). The VM executes the actual bytecode of smart contracts, but it does not manage account balances or message flows directly, because of its sandboxed environment. These responsibilities remain with the Executor that manages invocation of VM via vm.run() during the Compute phase.

The Compute Phase is the only stage where the TVM executes the smart contract code invoked by a message containing a *bag of cells*. All internal computations occur during this phase which are not involving other contracts states. However, the contract may invoke libraries provided by the Executor which are *stateless* smart contracts. The Compute Phase is deterministic because it involves underlying deterministic structures (*cells*), its result depends on the input data and the current state of the smart contract. If an exception occurs during the Compute Phase (e.g., due to out-of-gas errors), the entire transaction is aborted, and the Action Phase does not start. At the end of this phase, the TVM prepares a set of "output actions" for dispatch during the Action Phase. Besides exceptions that may occur as a result of smart-contract implementation errors, TVM also supports stopping and resuming smart-contracts via Continuations. In the TVM, they are first-class citizens and an equivalent to execution tokens, thanks to deterministic cells. They enable control flow graphs, and participate in exception handling of a smart contract.

In the Action Phase, the Executor dispatches output messages created during the Compute Phase. These actions may include calling other smart contracts within the blockchain network or leading to various outcomes such as token transfers and state changes of the receiving contract, and code change of the contract that just finished the Compute Phase. However, the actual state changes only occur if the Action Phase is successfully completed.

Tycho's Virtual Machine architecture represents an implementation of original TON VM principles [36] with practical enhancements for high-performance blockchain execution. The separation of concerns between the Executor and TVM creates a robust framework where the sandboxed virtual machine focuses purely on deterministic computation while the Executor manages the broader transaction lifecycle and system interactions.

Most importantly, the architecture's support for parallel execution of TVM instances positions Tycho to handle the demanding throughput requirements of modern blockchain applications.

Data Storage

All previously described architectural components, namely DAG mempool, Collator and Executor rely on a sophisticated multi-layered storage architecture (see diagram on Figure 10) that ensures persistence and high throughput of dependent sub-systems. The storage system is organized into several distinct RocksDB databases that serve different purposes:

- Base DB stores core blockchain data including blocks, states, cells, and archives;
- RPC DB stores transaction and account data optimized for RPC queries;
- Mempool DB stores DAG-related data like points and their statuses.

This way, the storage system creates multiple database instances with resource partitioning to avoid contention between different workloads. The storage system implements intelligent memory allocation based on available system memory. It calculates memory usage for various components and distributes the remaining memory between the cells cache and RocksDB LRU cache.

Serialization & Low-level Storage Optimization

At the low level, Tycho nodes store all data in a collection of cells, similar to the TON VM architecture. This collection is called a "bag of cells," which is, mathematically, a set that prevents cell duplication. This design transforms the main paradigm of "everything is a value" from the TVM whitepaper [31, 36] into an "everything is a cell" principle, supporting a flexible type system and more compact serialization formats. Each cell is a binary structure that can carry up to 1023 bits of data and up to 4 references to other cells. These cells can form deterministic DAG structures, which enable the construction of Merkle trees and support reference counting.

The system implements deduplication at multiple levels through a specialized cell database. At runtime, cells with identical hashes point to the same physical memory location, with lazy loading ensuring that entire trees aren't loaded until accessed. The underlying RocksDB storage maintains mappings of hash - > (refcount, child_hashes, data), where reference counting is crucial since individual cells can appear multiple times across the entire DAG.

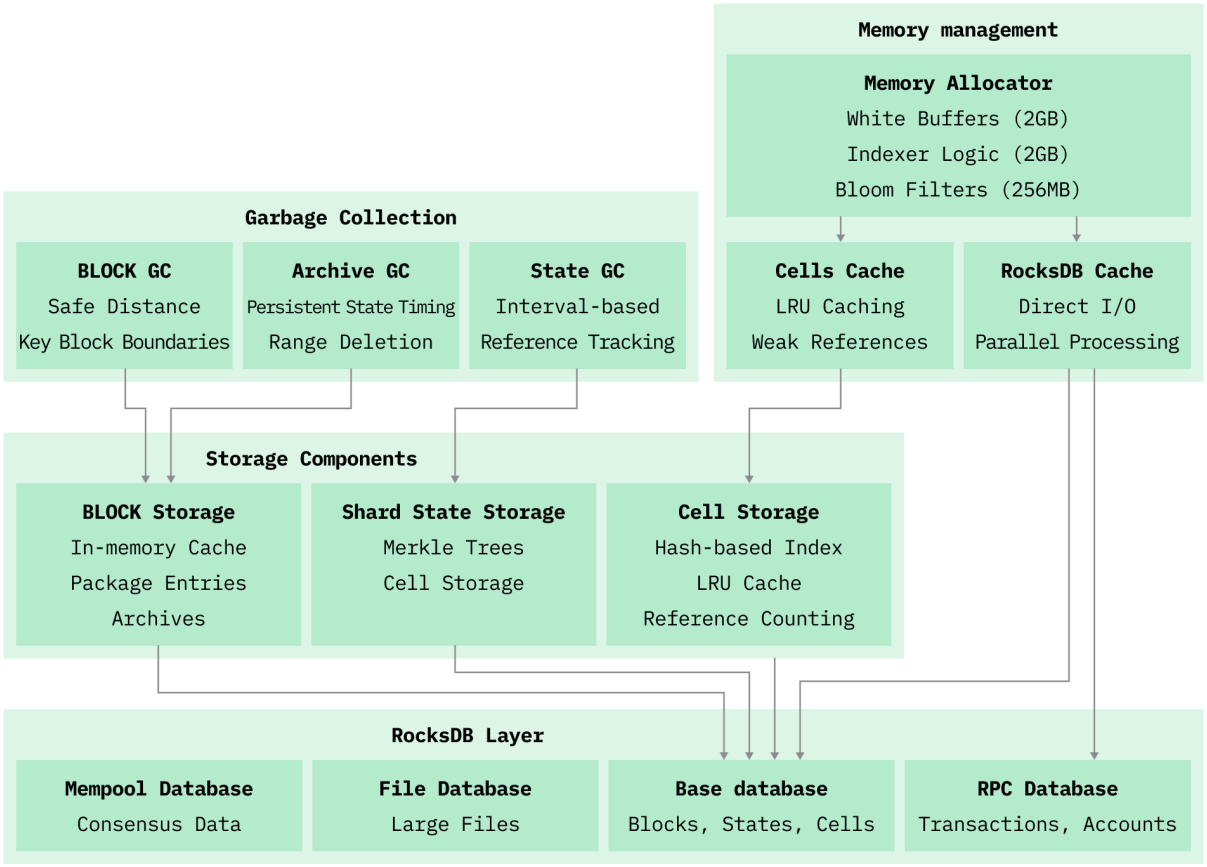

Figure 10. Tycho Node Storage System functional components.

As part of the Shard State Storage inside Base DB, the cells column family receives the most sophisticated optimization as it stores the core Merkle tree data. The hash-based memtable with buckets eliminates the insertion overhead of maintaining sorted order during the high-frequency cell write operations, while the large memtables with buffers provide substantial write buffering to reduce flush frequency and maintain consistent write performance. The 5-level LSM tree structure is specifically tuned for the large-scale dataset typical of blockchain cell storage, providing efficient compaction strategies that balance read and write amplification. Bloom filters offer high-precision negative lookups, crucial for quickly determining cell existence without expensive disk seeks, while direct I/O bypasses the kernel page cache to prevent double-buffering and gives RocksDB full control over memory allocation. Finally, compact-on-deletion ensures immediate space reclamation during garbage collection operations, preventing the accumulation of tombstones that would otherwise degrade read performance and storage efficiency. This is a critical consideration given the frequent state cleanup operations inherent in blockchain node operation.

Shard State Storage

The Shard State Storage in Tycho manages blockchain state data in coordination with block storage. Most importantly it contains the core storage for individual state cells with sophisticated optimization mentioned in the previous section. An auxiliary Temporary Cells table is used during state processing for intermediate cell storage.

Shard States Table connects to the Block storage via mapping BlockId to the root cell hash of the shard. The Block Handle Storage, a sub-component of the Block Storage, tracks state availability flags and ensures proper sequencing of block processing and state computation before feeding it into Shard States. The system implements coordinated garbage collection that removes outdated states while preserving referenced cells through atomic reference counting, maintaining storage efficiency as the blockchain grows.

The system maintains separation of concerns from block storage while sharing the same database infrastructure, allowing both systems to operate efficiently within the same node instance.

State Management and Merkle Updates

The Collator and its components facilitate state change for the whole Tycho protocol using several key data structures:

- WorkingState that contains the complete context including the usage tree, master chain data, and previous shard data;
- PrevData which encapsulates both observable states (with usage tracking) and pure states (original states);
- ActualState which is Core state maintained throughout all collation phases.

Block processing involves Merkle updates containing virtualized old (ActualState) and new states (WorkingState), where unchanged cells are pruned to retain only their hashes. When applying a new block, the system performs a depth-first traversal from the root, stopping when encountering existing cells. All traversed cells have their reference counts incremented by one, with all key-value pairs written to RocksDB in a single transaction using merge operators.

Previous states undergo garbage collection through similar traversal, decrementing reference counts. RocksDB's compaction filter removes cells with zero reference counts. The system addresses performance challenges through parallel graph

traversal for large states and optimized I/O operations. However, state update complexity scales as O(updates × $\log_2$(state_size)), creating throughput limitations depending on the amount of active accounts.

State sharding presents another challenge. While reducing individual shard update sizes through independent processing ($\log_2$(size/n)), physical separation into different databases eliminates deduplication benefits, potentially doubling storage requirements. Future optimizations include implementing cuckoo maps for faster cell existence checks and developing custom databases optimized for hashmap operations with counter increments and write-ahead logging.

Previously described Work units (WUs) system help to coordinate the state management system by means of measurement of computational resource consumption across different phases of block collation. This includes tracking work units for message preparation, execution, and finalization phases. The work unit tracking feeds into the state management system by updating the field in the new observable state, providing feedback for future collation attempts and resource management.

Block Data

Tycho's block storage system implements a multi-tiered architecture that balances performance, storage efficiency, and data availability. Unlike simpler blockchain implementations that store blocks in flat file systems (for example Bitcoin's) or basic key-value stores, Tycho employs a comprehensive caching hierarchy with TTL-based in-memory caches, immediate package storage for critical block components, and intelligent data splitting for large blocks. This design mirrors and extends TON's storage philosophy but introduces more granular control over data lifecycle management and enhanced compression strategies optimized for different data types.

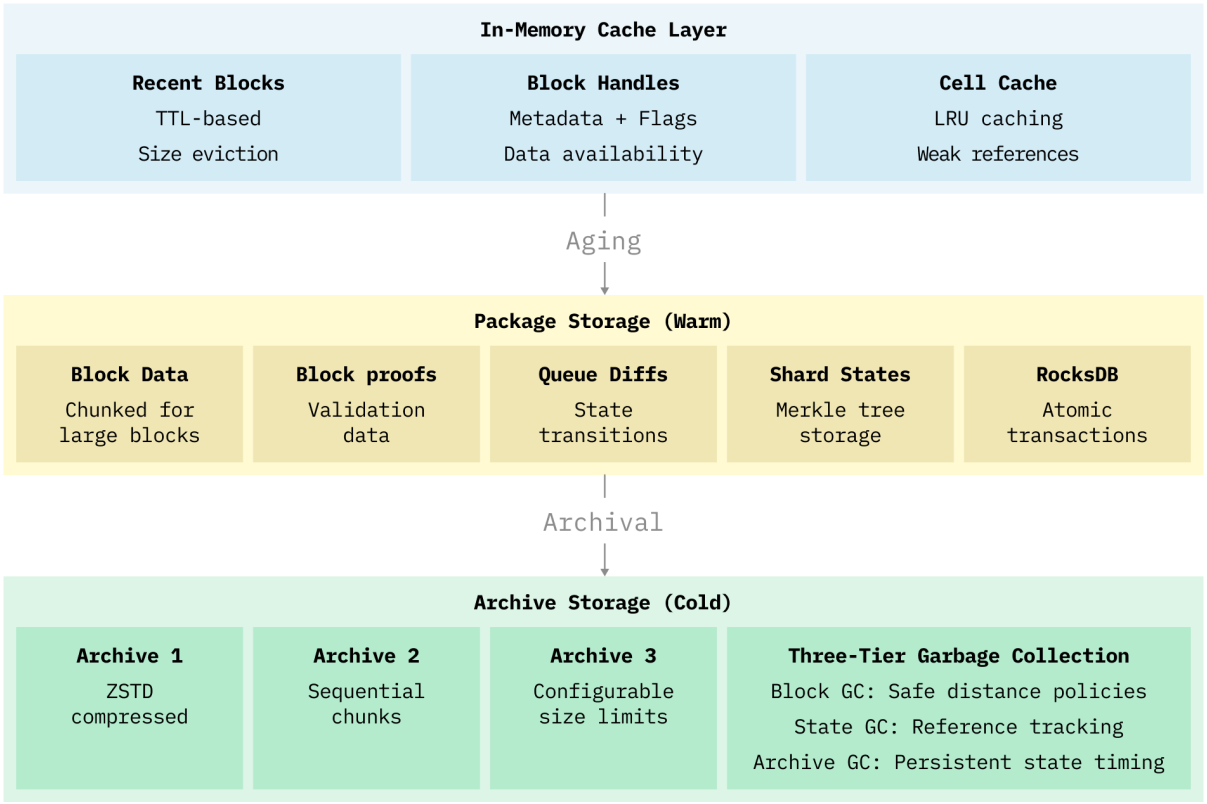

Figure 11. Tycho Block Storage Architecture.

The protocol implements a three-tier storage model consisting of hot in-memory caches, warm package storage, and cold compressed archives (Figure 11). Recent blocks remain in fast-access storage with full metadata availability, while older blocks are progressively moved to ZSTD-compressed archives organized as sequential chunks with configurable size limits. This archival approach significantly reduces storage overhead compared to traditional blockchain nodes that maintain all historical data in uniform format. The block handle system provides efficient metadata tracking through bit flags that indicate data availability status. There is no distinction whether block data, computed state, proofs, or queue diffs are stored, thus enabling nodes to quickly determine what information is accessible without expensive disk operations.

Tycho's block storage tightly integrates with its sophisticated shard state storage system. This integration allows for efficient state transitions where new blocks can reference existing Merkle tree cells from previous states, similar to TON's approach but with enhanced garbage collection mechanisms. The system implements a three-tier garbage collection architecture that independently manages blocks, states, and archives, ensuring optimal storage utilization while maintaining data integrity and availability guarantees for active network participants.

The storage system incorporates several advanced optimization techniques that distinguish it from conventional blockchain architectures. Extensive use of Bloom

filters optimizes memory usage for cell lookups, while hash-based indexing enables efficient point lookups and binary search indexing supports range queries. The system implements adaptive write optimization with pipelined writes and rate limiting with auto-tuning capabilities. Unlike TON's more uniform compression approach, Tycho employs differentiated compression strategies. It leverages ZSTD compression for compressible transaction and metadata and no compression for already-compressed archives and cell data. For values exceeding 32KB thresholds it uses specialized blob storage. These measures altogether ensure optimal storage efficiency across diverse data types while maintaining high-performance access patterns for active network operations.

Scalability & Interoperability

How Tycho Achieves Better Scalability

Queue State Separation from Shard State

The Tycho follows a principle of smart-contract state organization which includes storing the output message queue of the related account as it was mentioned in TON Whitepaper (sections 2.3.20, 2.4.7 in [31]). The approach that was chosen for implementation in TON allows storing the outgoing queue as a hashmap within the shard state. But when queues become large, these hashmaps grow significantly, causing noticeable slowdowns in both queue operations and overall state management. Tycho solves this by separating outgoing queues entirely from the shard state and storing them directly in the RocksDB table ordered by logical time hash with the key structure:

`{partition_id}{src_shard}{lt}{hash}`

This approach provides several advantages:

- **Faster reading and parsing.** Messages with metadata are serialized and stored entirely in binary form, rather than requiring cell tree traversal
- **Efficient filtering.** Only the first few bits need to be read to check recipient addresses without full message parsing.
- **Linear queues are significantly faster** to read, trim from the bottom, and append to the top compared to hashmap operations.
- **Optimized iteration.** RocksDB iterators enable sequential reading of outgoing messages from source shards within specified logical time hash ranges.
- **Removing large queue hashmaps** from shard state improves state management performance.

The use of queue diffs for state changes is equally important - containing pre-ordered new messages that were created but not executed during block collation, along with processing boundaries. Peer nodes exchange queue diffs without messages content which saves on bandwidth and traffic. During synchronization, the node that catches up with the network, syncs only the message index and obtains message content from outgoing messages in the past blocks.

High Performance Message Grouping

The parallel transaction execution optimization addresses a fundamental bottleneck in blockchain transaction processing. The goal is to execute Na*Nm transactions in groups where Na is the number of unique accounts and Nm is sequential messages per account.

The multi-range reading mechanism with separate readers for different ranges allows parallel execution of messages from subsequent blocks alongside those from previous blocks, while maintaining strict ordering guarantees per account. By reading more messages into buffers and then optimally filling groups from those buffers, the system avoids scenarios where overloaded accounts force execution of underfilled groups.

Additionally, Tycho prevents large queues on single accounts from monopolizing execution capacity through a sophisticated partitioning system. When message count on account A1 exceeds limits, it's moved to isolated partition 1, where messages execute with lower priority but guaranteed minimal throughput. The routing mechanism using cumulative statistics and the <account_id, partition_id> mapping in diffs ensures deterministic assignment while preventing "bouncing" between partitions at threshold boundaries.

Minimized Data to Validate

The deterministic collation guarantee enables incredible efficiency - since DAG at mempool level guarantees external message ordering and message grouping is strictly deterministic, all consensus nodes produce identical blocks. As it was previously pointed out in the section about Collator validation only requires node public key and signature of block root cell hash. This eliminates the need to transmit actual blocks during validation, with signature exchange completing very quickly over the network.

These architectural decisions represent a fundamental rethinking of blockchain collation that goes far beyond incremental optimizations - they're the core innovations that enable the performance gains over TON's approach.

Layer-2 Scaling

With regard to Tycho, detailed L2 plans are not currently finalized. At the moment rollups may be considered as the most effective and well tested way to introduce scaling in account-based blockchains. Whereas zero-knowledge proof schemes were considered more optimal for TVM design, they are not efficient and involve massive overhead costs for the VM.

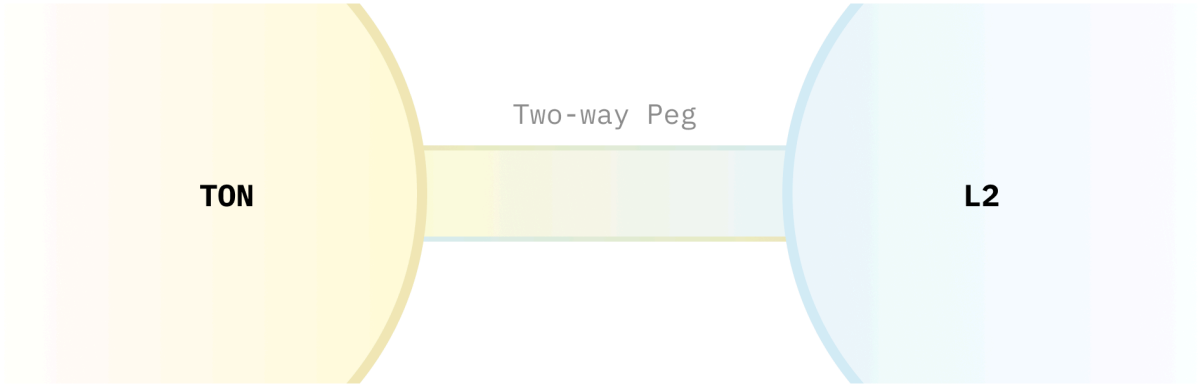

Figure 12. Cross-chain contracts facilitate bi-directional transfers between TON and a Tycho-based L2 chain

Ongoing work focuses on implementing compatible chains that can efficiently interact with each other via dedicated contracts and intermediate light clients (see scheme on Figure 12). The light client serves as an oracle, providing block proofs for cross-chain contracts while validating structures and maintaining validator set transitions. Together, these components eliminate traditional bridge elements like relays or third-party signers, with security assumptions reduced to regular on-chain transfers with additional time-based limitations. The oracle provides only data and proofs without participating in consensus. Anyone can deploy the oracle service and use existing contracts, or manually submit blocks in case of oracle failure.

Tycho-based chains are supposed to serve specific applications, such as Decentralized Exchange (DEX) protocols, which face ever-increasing demand for capacity in terms of gas, storage, and TPS. In this architecture, a single native base layer token becomes a L2 token via cross-chain contracts with 1:1 backing. For other tokens, compatible token standards are being developed to enable efficient bridging with low maintenance costs.

Tycho-Based Protocols and TON

The Tycho prototype inherited a high degree of compatibility with TON regarding basic concepts and architecture. In contemporary design, compatibility is not an end goal but rather a means for achieving better interoperability with TON nodes and the protocol in general.

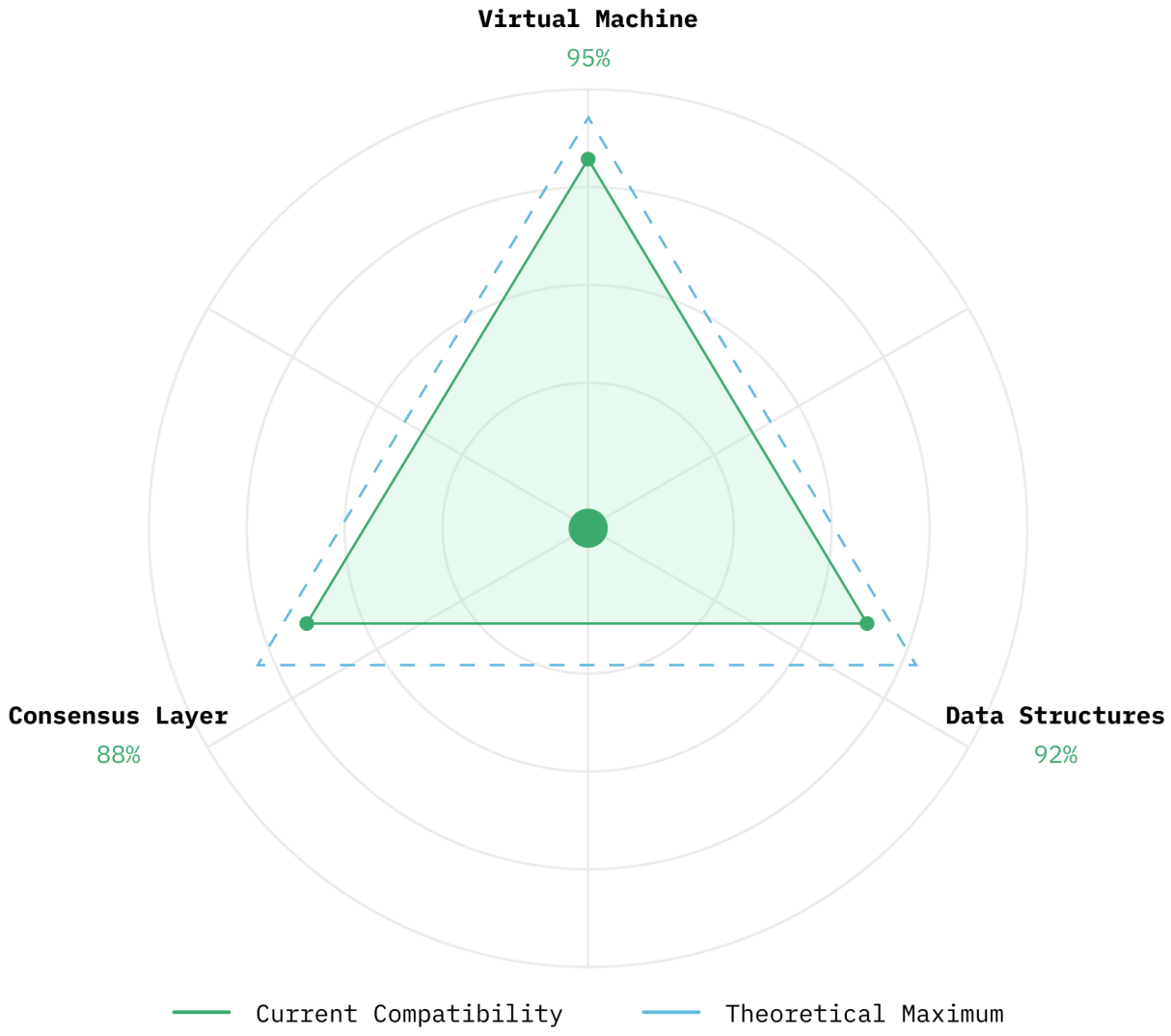

Figure 13. Tycho-TON Compatibility Assessment

The visualization on Figure 13 represents common dimensions of comparison that serve as simplified indicators of the overall compatibility score. The most significant contributors to the high overall score are shared data structures and TON Virtual Machine integration. These foundational elements allow developers to build a cohesive ecosystem of smart contracts that are fully compatible with each other, serving as the basis for more advanced approaches to achieve complete interoperability.

Furthermore, unification of the election system through the same opcodes and structures enables the construction of "trustless bridges," assuming that the trust model is no worse than that of the base layer of a single protocol.

Tycho nodes feature a dedicated component designed to provide a seamless experience when switching between networks for end users. The TonCenter API

delivers a TON node RPC API with TON-compatible data structures and responses that can be accessed by third-party applications without requiring any changes or incurring maintenance costs.

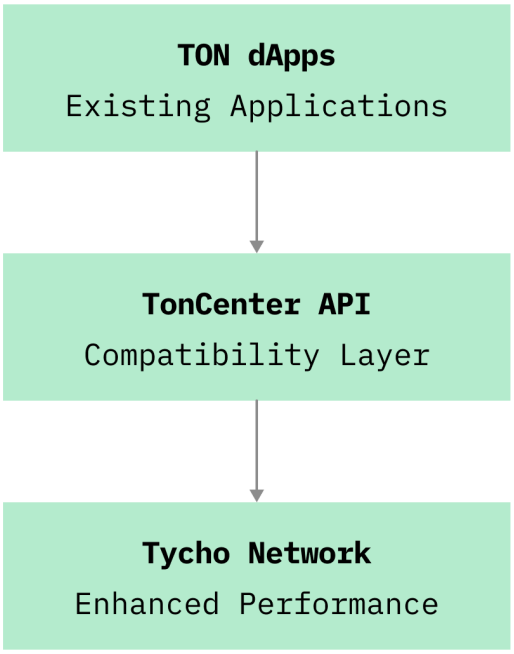

Figure 14. Application-level compatibility

Network

QUIC (Quick UDP Internet Connection) serves as a primary transport protocol for peer-to-peer communication between Tycho blockchain nodes in place of ADNL (Abstract Data Network Layer) and RLDP (Reliable Large Data Protocol) used as network communication protocols in TON. The network layer is built around an established library to provide reliable, multiplexed communication over UDP. In addition to that, If compared to TON ground up approach, Tycho networking layer focuses rather on direct peer-to-peer connections than on complex routing topologies and connection management handled by the underlying QUIC implementation. Besides, while ADNL is also built on UDP, QUIC has better deliverability assurances than ADNL and RLDP which runs on top of ADNL.

Similarly to TON ([31], section 3.2), Tycho uses a DHT (Distributed Hash Table) for peer discovery (schematically represented by Figure 15). The implementation is a Kademlia-like system that provides both peer discovery and distributed data storage capabilities. Each peer has a 256-bit ID, and peers are organized using XOR distance metrics. The closer two peer IDs are in XOR distance, the more likely they are to know about each other. Both systems use Ed25519 cryptography for peer identity, but handle it differently. TON maintains both full and short node IDs with complex address list management and version tracking. Tycho uses a simpler PeerId wrapper around Ed25519 public keys, relying on TLS certificate verification for authentication.

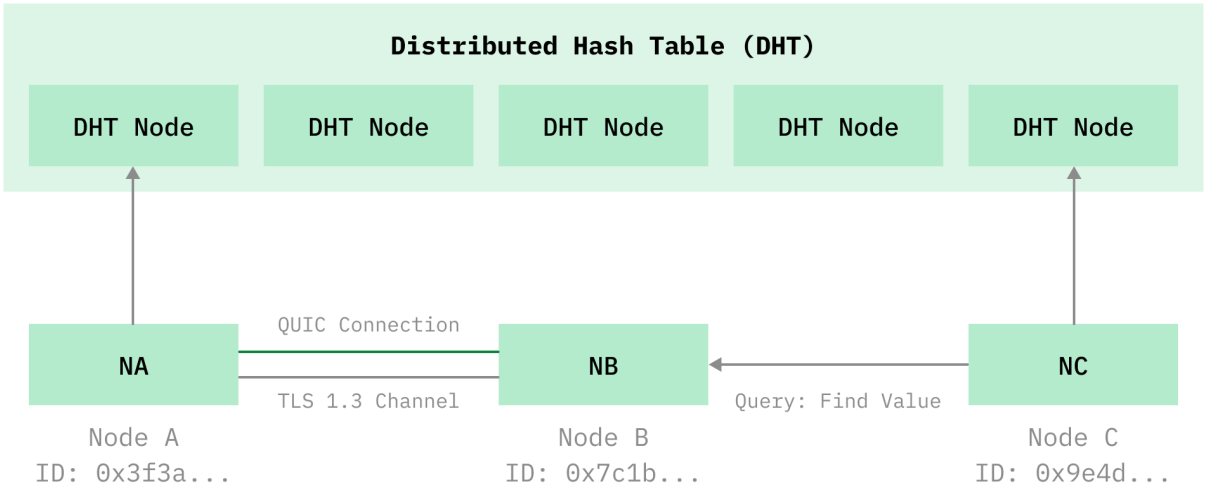

Figure 15. A schematic view of Tycho nodes interacting via QUIC with DHT based peer discovery.

Beyond basic DHT discovery, Tycho supports two types of overlay networks that operate on top of the base DHT infrastructure. Public overlays are topic-based networks where peers automatically discover others who share interest in the same

overlay topic - when a peer joins a public overlay (identified by a unique overlay ID), it periodically announces its participation in the DHT and actively searches for other peers who have announced participation in the same overlay, creating a self-organizing network of peers with shared interests. Private overlays, in contrast, are closed networks with explicitly managed membership lists where peer discovery is manual rather than automatic. In these kinds of overlays, peers maintain a predefined list of member peer IDs and use the DHT solely to resolve the current network addresses of these known members, ensuring that only authorized peers can participate while still leveraging the DHT's address resolution capabilities to handle dynamic IP addresses and network changes.

Benchmarking

Top-Level Performance Measurements

The Tycho protocol is undergoing active development with a rapidly evolving codebase and currently lacks convenient tools for formal protocol auditing. For illustrative purposes, a public website displaying protocol statistics can be used as a trusted source of various performance metrics [32]. Additionally, early adopters such as Hamsterchain have reported throughput levels reaching approximately 35,000 TPS in production environments, while internal testing demonstrates results up to 140,000 TPS.

The network can be thoroughly stress tested using a specialized toolkit composed of two complementary components designed to simulate realistic network load conditions. In this section, we provide a description of the main benchmarks and report them alongside publicly available data on other protocols.

The data used in this analysis were drawn from official documentation, academic papers, and technical whitepapers of the respective protocols [5,9,12,15,17,19,20,21]. The cited sources provide approximate values for transaction throughput (TPS) and finality latency under nominal or best-effort conditions. In case of reported multiple values, we adopted median or typical estimates to ensure consistency and collected them in Table 1, which served as the basis for the plot in Figure 16.

Table 1.
Reported Performance of DAG-Based Blockchain Protocols

Protocol	Throughput (TPS)	Latency (s)
IOTA	~800	20
Avalanche	~4,500	2
Hashgraph (Hedera)	~10,000	4
PHANTOM/GHOSTDAG	~400	60
BlockDAG	~400	45
Mysticeti	~100,000	1
Sui/Aptos (Shoal++)	~130,000	1.5

The Tycho main benchmark setup consists of the primary utility, *nekroddos* [38], that serves as a load generator capable of creating diverse network stress patterns under varying scenarios, and the secondary tool that provides mass deployment of typical wallet contracts for enlarging chain state. Nekroddos supports multiple load generation modes, enabling developers and operators to simulate real-world conditions, such as high transaction volumes, burst traffic, and sustained loads. The tool interacts with pre-deployed smart contracts on the target network, ensuring authentic transaction patterns rather than synthetic test loads.

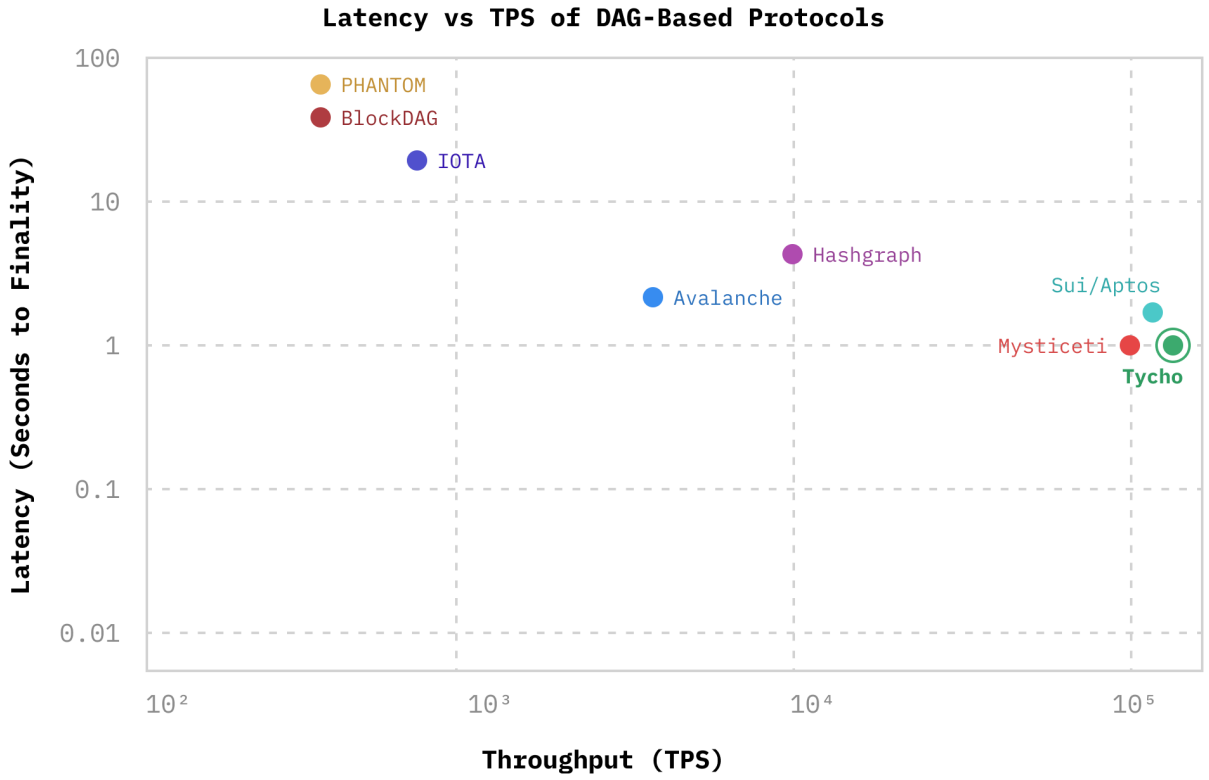

Figure 16. Estimated Throughput vs Finality Latency for DAG-Based Protocols

To support the nekroddos utility, a dedicated *wallet deployer* component manages the initial setup by deploying necessary smart contracts to the Tycho network. This tool automates contract installation and generates a comprehensive list of contract addresses that serve as input for the stress testing utility. It ensures a properly configured environment with all dependencies required for accurate, production-like testing.

Reported results in tests conducted using both utilities demonstrated a peak performance of 140,000 TPS for external messages, with an average latency of 1.009 seconds. These results are plotted alongside other DAG-based protocols in Figure 15.

The protocol constantly improves and further research in developing reliable benchmarks for cross-protocol comparisons is needed. For example, a major factor

behind the observed throughput growth up to 130,000 TPS is the size of the chain state, i.e. the total number of active accounts. While this factor is not taken into consideration in publications about competing DAG implementations, it plays a significant role in Tycho's performance profile. However, the only metrics that users do care about is simply throughput and settlement speed. Thus, while internal benchmarking may be considered a mission-critical feedback mechanism, on the top level only aggregated parameters help to deliver comprehensive inputs for making informed decisions.

Conclusion

DAG-based blockchain protocols represent a promising yet evolving design space that seeks to overcome performance and scalability limitations of sequential systems considered as legacy systems nowadays. In this paper, we outlined major design features of the newly proposed Tycho protocol.

Tycho demonstrates excellent results in terms of throughput and latency, in line with what already existing protocols demonstrate in reports. Key challenges remain in balancing performance with decentralization, ensuring consistency under concurrency, and achieving robust finality in permissionless settings. While different Tycho-based protocols may expose different trade-offs, the underlying architecture remains highly compatible and expected to deliver superior interoperability with optional implementation of so-called “layer two” protocols on top of base layer.

Further research is required to formalize benchmarking practices, understand adversarial resilience, and evaluate the applicability of DAG-based models to general-purpose smart contract execution. Initial results demonstrate satisfactory throughput of the Tycho protocol, which is trending toward such high-performance, low-latency protocols as Aptos and Sui.

References

[1] Thulasiraman, K. and Swamy, M. N. S. (1992). Graphs: Theory and Algorithms. John Wiley & Sons.

[2] Bang-Jensen, Jørgen and Gutin, Gregory (2008). Digraphs: Theory, Algorithms and Applications. Springer.

[3] Christofides, Nicos (1975). Graph Theory: An Algorithmic Approach. Academic Press.

[4] Spiegelman, Alexander, Giridharan, Neil, Sonnino, Alberto, and Kokoris-Kogias, Lefteris (2022). Bullshark: DAG BFT Protocols Made Practical. Available at: https://sonnino.com/papers/bullshark.pdf

[5] Shao, Da, van Renesse, Robbert, and Sirer, Emin Gün (2023). Mysticeti: Leaderless BFT with Deterministic Commit. Available at: https://arxiv.org/abs/2310.14821

[6] Danezis, George, Kokoris-Kogias, Lefteris, Sonnino, Alberto, and Spiegelman, Alexander (2022). Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. In Proceedings of the Seventeenth European Conference on Computer Systems, ACM.

[7] Bagaria, Vivek, Kitajima, Yusuke, Zhou, Yi, Das, Sankalp, Stoelinga, Mariëlle, and others (2024). Reusable Formal Verification of DAG-based Consensus Protocols. Available at: https://arxiv.org/pdf/2407.02167

[8] Broxus (2025). Tycho: Reference Implementation of Tycho Protocol - Consensus Module. GitHub repository. Available at: https://github.com/broxus/tycho/tree/master/consensus

[9] IOTA Foundation (2020). The Coordicide. Available at: https://files.iota.org/papers/20200120_Coordicide_WP.pdf

[10] Saa, Olivia, Cullen, Andrew, and Vigneri, Luigi (2023). IOTA 2.0 Incentives and Tokenomics Whitepaper. Available at: https://files.iota.org/papers/IOTA_2.0_Incentives_And_Tokenomics_Whitepaper.pdf

[11] IOTA Foundation (2023). IOTA Ecosystem DLT Foundation White Paper. Available at: https://files.iota.org/dlt/White_Paper_IOTA_Ecosystem_DLT_Foundation.pdf

[12] Team Rocket (2019). Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. Available at: https://arxiv.org/abs/1906.08936

[13] Amores-Sesar, Ignacio, Cachin, Christian, and Schneider, Philipp (2024). An Analysis of Avalanche Consensus. arXiv preprint arXiv:2401.02811. Available at: https://arxiv.org/abs/2401.02811

[14] Ava Labs (2020). Avalanche Consensus Protocol Whitepaper. Available at: https://www.avalabs.org/whitepapers

[15] Baird, Leemon and Hedera (2023). Hedera Whitepaper v2.2. Available at: https://files.hedera.com/hh_whitepaper_v2.2-20230918.pdf

[16] Hedera (2023). Hedera Consensus Service Whitepaper. Available at: https://files.hedera.com/hh-consensus-service-whitepaper.pdf

[17] Sompolinsky, Yonatan, Wyborski, Shai, and Zohar, Aviv (2018). PHANTOM: A Scalable BlockDAG Protocol. Available at: https://eprint.iacr.org/2018/104.pdf

[18] Perešíni, Martin, Benčić, Federico Matteo, Malinka, Kamil, and Homoliak, Ivan (2021). DAG-Oriented Protocols PHANTOM and GHOSTDAG under Incentive Attack via Transaction Selection Strategy. arXiv preprint arXiv:2109.01102. Available at: https://arxiv.org/abs/2109.01102

[19] DAG Systems Ltd (2024). BlockDAG Whitepaper. Available at: https://blockdag.network/blockdag-new-whitepaper.pdf

[20] Mysten Labs (2022). The Sui Smart Contracts Platform. Available at: https://docs.sui.io/paper/sui.pdf

[21] Aptos Labs (2022). The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure. Available at: https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf

[22] Electric Capital, "2024 Crypto Developer Report," 2024. Available: https://www.developerreport.com/

[23] A. Falcão, "Protocol-based smart contract generation," 2023. Available: https://afalcao.dev/

[24] Tezos Foundation, "Tezos architecture," 2023. Available: https://docs.tezos.com/architecture

[25] J. Chen and S. Micali, "Algorand," 2017. Available: https://www.algorand.com/

[26] M. Bartoletti and L. Galletta, "A formal model of Algorand smart contracts," in *Proceedings of the 2nd Workshop on Trusted Smart Contracts*, 2018.

[27] S. Blackshear, D. Costanzo, and G. Nelson, "Resources: A safe language abstraction for money," in *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 2019.

[28] M. Patrignani, "Robust safety for Move," in *Proceedings of the 2020 ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2020.

[29] K. Zandberg and E. Baccelli, "Minimal virtual machines on IoT microcontrollers: The case of Berkeley Packet Filters," in *Proceedings of the 2020 International Conference on Embedded Wireless Systems and Networks*, 2020.

[30] Nikolai Durov (2020). Catchain Consensus: An Outline. Available at: https://api.semanticscholar.org/CorpusID:212700976

[31] TON Network, "TON Whitepaper," 2020. Available: https://ton.org/whitepaper.pdf

[32] Corbett, J. C. et al. (2013). Spanner: Google's globally-distributed database, ACM Transactions on Computer Systems, 31(3), Article 8. Available: https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi2012.pdf

[33] Narula, N., Catalini, C., Duca, M., Foley, S. N., Lipton, A., Niepelt, D., et al. (2022). A high performance payment processing system designed for central bank digital currencies (OpenCBDC). MIT Digital Currency Initiative and Federal Reserve Bank of Boston. Available: https://www.bostonfed.org/-/media/Documents/project-hamilton/2022/Project-Hamilton-Phase-1-Technical-Paper.pdf

[34] Broxus, "Tycho," 2023. Available: https://github.com/broxus/tycho

[35] "Tycho VM," 2023. Available: https://github.com/broxus/tycho-vm

[36] N. Durov, "Telegram Open Network Virtual Machine," 2020. Available: https://ton.org/tvm.pdf

[37] Tycho Protocol Team: Tycho protocol: Boosted TVM blockchain protocol (2025), https://tychoprotocol.com, build FAST L1/L2 TVM networks based on state of the art DAG consensus

[38] KappaShilaff: nekroddos: Load generation utility for tycho network stress testing (2024), https://github.com/KappaShilaff/nekroddos/tree/master