TAC Whitepaper

Abstract

TAC is an EVM for Telegram: a Layer 1 with a TON (The Open Network)-specific CrossChain Layer, that connects Ethereum dApps and developers with Telegram's users without exposing users to bridges, extra wallets, or wrapped assets. Today, EVM dApps that seek Telegram's billion-user reach must either rewrite their code for TON or accept fragmented liquidity and compromised UX. TAC eliminates this gap through the concept of **Hybrid dApps -** EVM dApps natively accessible by any TON wallet holder and working with TON assets, enabled by:

- **TAC EVM Layer** — a CosmosSDK + Ethermint-based blockchain that runs unmodified Solidity contracts, achieves ≈2-second finality through dPoS
- **TON Adapter** — a bidirectional, asynchronous messaging layer designed as a collateral-backed sequencer network, turning every deployment into a Hybrid dApp where transactions start and end natively on TON.
- **TAC SDK** — a developer toolkit that leverages the TON Adapter to build TON-native frontends with minimal effort, including Telegram MiniApps, bringing EVM-powered logic directly inside the Telegram environment.

The key features of TAC's architecture and **Hybrid dApps** framework are:

- TON and Telegram users can enjoy the advancements of EVM ecosystem with its blue chip dapps, deep liquidity and use cases variety in seamless way
- Streamlined access for the world largest and most professional developer community to build in TON ecosystem using the same
- Standard Layer1 value accrual mechanism for TAC token and its economic alignment with TON

TAC end game is creating a new business model for Consumer Telegram MiniApps, with tens of millions of users, that will play the role of distributing DeFi primitives (swaps, borrow, yield) to their user base.

Table of contents

TAC Whitepaper
- Abstract
- Table of contents
- 1. Problem Statement & Background
- 2. Architecture Overview
  - Guarantees
  - TAC Components
    The subsequent sections dive deeper into each of these stages, clearly highlighting the specific role of core architectural components: Proxy Applications, TON Adapter, Sequencer Network, Executor mechanism, and TAC EVM Layer.
  - 1. SDK
  - 2. Cross-Chain Messaging
  - 3. Security, sequencing and consensus
  - 4. EVM-side Proxy Contract
  - 5. Executing
  - 6. Fee Settlement
  - Directional Fee Logic
  - Fee Structure
  - Gas Pricing Model (EIP-1559 Compatible)
  - User-Side Fee Estimation
  - Executor-Side Fee Verification
  - Fee Fixation
  - Risk Mitigation
  - User Interaction Flow
  - Reentrancy Protection and Message Uniqueness
  - Economic Security and Incentives
  - Future Security Enhancements
  - Planned FROST Consensus Upgrade
- 3. TAC Token and Tokenomics
  - Tokenomics
- 4. Governance and Upgradeability
  - Governance Scope
- 5. Use Cases & Applications(template)
  - 5.1 TON-Originated DeFi Interactions
  - 5.2 Hybrid Telegram MiniApps
  - 5.3 Cross-Chain dApps and Proxies
  - 5.4 Automation and Delegation
  - 5.5 Future Integrations
- 6. Roadmap
- Conclusion

1. Problem Statement & Background

Despite its rapid adoption, **The Open Network (TON)** lacks native access to the EVM ecosystem — home of the largest blue-chip DeFi protocols and widely adopted developer tooling. At the same time, EVM applications have no direct path to Telegram's billion-user audience without relying on UX sophistication which would barely work for mass users - complex bridges, wrapped assets, and multi-wallet flows. Even with these UX complexities and security trade-offs existing solutions rather extract value from the TON ecosystem. Another important point is that with its separated Fun-c developer environment creating a large-scale builders community is a complex and a very time consuming task for TON.

These limitations introduce significant frictions:

- TON DeFi and consumer apps ecosystem struggle to unlock its full potential in terms of TVL, liquidity depth, assets variety, quality and utilization as well as use cases variety (especially for yields)
- Users must manage separate wallets (TON and EVM) and handle token conversions manually
- dApps must find a way to re-deploy in a totally different environment and start there from scratch
- UX for Telegram MiniApps and mobile-first users suffers due to fragmented infrastructure

TAC is built to resolve these issues. It is a Layer-1 blockchain that connects TON and EVM. Users can interact with EVM dApps directly from their TON wallets—no manual asset bridging, no browser extensions, and no additional wallets required:

- EVM Ecosystem with its full capabilities is unlocked for TON and Telegram users
- Users can access dApps with one wallet, one asset, and no need to bridge assets
- Developers write once in Solidity and reach a billion Telegram users - largest developer community meets largest user base
- MiniApps integrate Web3 features—like payments, swaps, lending—without external SDKs;

TAC is not a cross-chain workaround—it is a **native execution path** between two ecosystems that were never meant to connect. By removing barriers instead of building over them, TAC aligns UX, liquidity, and developer incentives across TON and EVM for the first time.

2. Architecture Overview

TAC is a Layer-1 blockchain with an cross-chain interoperability layer that enables interaction between asynchronous TON and Ethereum-compatible smart contracts. Its architecture is designed to support secure and deterministic cross-chain communication while preserving native user experience on the TON side. The protocol relies on off-chain batching, standardized message structures, and on-chain consensus to execute contract-level operations across both networks.

TAC's core components include:

TAC EVM Layer
TAC's Ethereum-compatible layer (**TAC EVM Layer**) is a fully-fledged Layer-1 blockchain environment based on Cosmos SDK and Ethermint technology. It provides a secure and scalable execution environment specifically optimized for Ethereum-compatible smart contracts, allowing existing EVM dApps to be deployed without modifications or code rewrites.

Key features of the TAC EVM Layer include:

- **Full EVM Equivalence:**
  Supports all standard EVM opcodes and is fully compatible with Ethereum tools and Solidity contracts. Developers can reuse existing contracts and tools (e.g., Remix, Hardhat, Truffle) without any additional effort.
- **Cosmos SDK and Ethermint Integration:**
  Built using battle-tested Cosmos SDK and Ethermint, TAC's EVM layer leverages the Tendermint consensus mechanism. This provides deterministic execution, fast finality (~2 seconds per block), and high throughput.
- **Delegated Proof of Stake (dPoS):**
  Secured by a Delegated Proof-of-Stake consensus model, allowing validators to stake tokens and users to delegate, ensuring economic security and validator alignment.
- **Transaction and Gas Model:**
  Fully supports EIP-1559-compatible gas pricing with base fees and priority fees, ensuring predictable and fair pricing for transaction execution.

TON Adapter
A mirrored pair of smart contracts deployed on both the TON and TAC EVM chains. The adapter manages the lifecycle of cross-chain messages, including locking and minting of assets, formatting of transaction data, verification of Merkle roots, and prevention of message reentrancy. The adapter is responsible for validating whether

messages are executable, rejecting invalid payloads, and enabling rollback in case of failed execution.

Sequencer Network
TAC's cross-chain messaging is coordinated by a decentralized network of off-chain sequencers. It starts with a permissioned decentralization (3/5 consensus) and will go fully decentralized. These nodes monitor events across both TON and EVM environments, aggregate them into Merkle trees, and submit roots for on-chain consensus.

Execution layer
An executor is selected for each cross-chain message, from the available options in the network. This can be either a Hybrid dApp executor or the user's own or any executor - the process is permissionless. Only the assigned executor is authorized to process the message and claim the reward. The operation is secured through cryptographic mechanisms, eliminating the need for executors to provide collateral.

On the TON side, the Adapter performs several essential tasks to securely manage cross-chain messages:

- **Message Reception and Validation**
  Upon receiving the message from the SDK, the TON Adapter on cross-chain layer smart contract verifies the correctness and consistency of the transaction structure, ensuring compatibility with TAC's cross-chain message standards.
- **Asset Validation and Locking Assurance**
  It rigorously validates asset transfers, ensuring the assets (e.g., Jetton tokens) are correctly locked and match the stated amounts and transaction parameters. This prevents inconsistencies and potential misuse of tokens during cross-chain operations.
- **Reentrancy Protection**
  Each message is assigned a unique timestamp and identifier, effectively preventing reentrancy attacks or duplicate transactions. This ensures that each user's transaction is executed exactly once, maintaining transaction atomicity.
- **Emitting Canonical Cross-Chain Messages**
  After validating the transaction and associated asset transfers, the TON Adapter formats the data into a canonical cross-chain message structure. This standardized message includes essential metadata such as timestamps, method calls, asset details, and user information.

Finally, the Cross-chain layer smart contract emits this structured and fully validated message to the **Sequencer Network**. Off-chain sequencers will then pick up this message, indexing it and preparing it for inclusion in Merkle trees, which will undergo decentralized consensus and subsequent execution on the TAC EVM Layer.

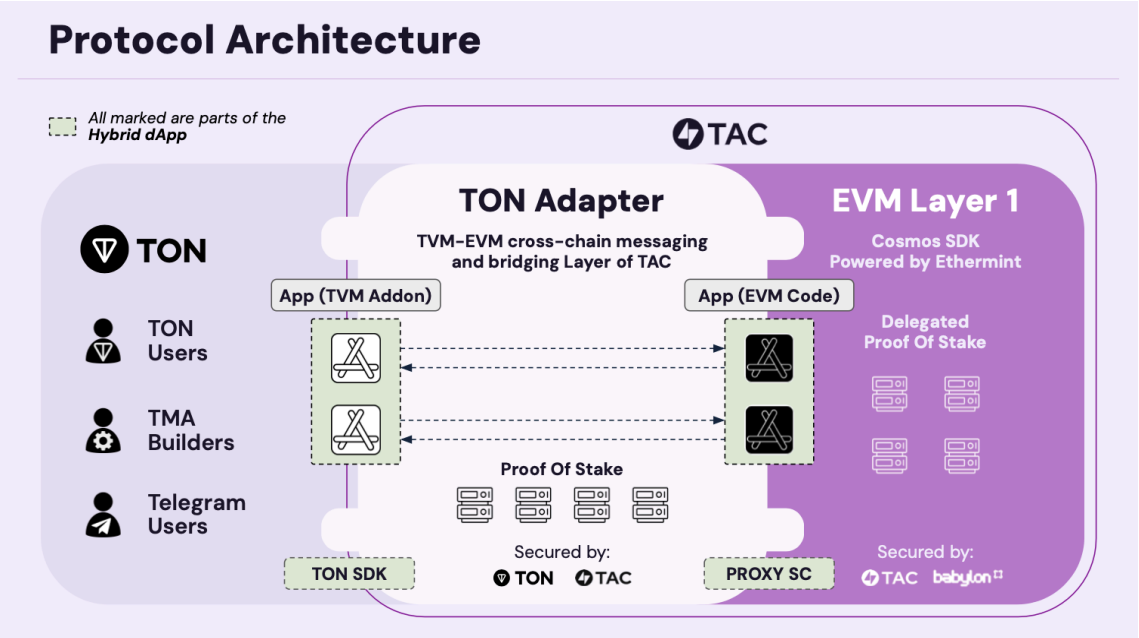
TON Adapter key attributes:

- **The TAC SDK:** Enables hybrid dApps that let TON users interact with EVM smart contracts without managing multiple wallets or dealing with cross-chain complexity.
- **Asset Validation:** Ensures assets are securely locked, matched, and prepared for cross-chain operations.
- **Reentrancy Prevention:** Assigns unique identifiers and timestamps to maintain transaction uniqueness.
- **Cross-chain Standardization:** Produces canonical messages fully compatible with TAC's decentralized sequencing and execution logic.

By integrating these critical tasks, the TON Adapter significantly enhances security, ensures deterministic cross-chain messaging, and provides a robust foundation for seamless integration of Hybrid dApps into the TON ecosystem.

Proxy Applications

Each application integrated with TAC is associated with two components - TAK SDK and EVM Proxy. The TAC SDK allows dApps to easily interact with cross-chain layer smart contracts. It encapsulates the correct formation of message parameters and their submission to the TON blockchain. The EVM-side proxy is a Solidity contract that receives validated input from the adapter and performs the specified method call on the target application. Proxy logic is standardized and does not require state maintenance.



Guarantees

TAC's architecture ensures deterministic finality, cross-chain atomicity, and a consistent execution model. All interactions are validated by sequencers, traceable via Merkle proofs, and confirmed through on-chain voting. Proxy contracts are stateless and generated automatically to reduce integration overhead. The protocol is designed for upgradeability, with clearly defined interfaces and consensus parameters managed via governance.

This structure enables secure and transparent interoperability between TON and EVM, while minimizing surface area for user error and implementation inconsistency.

TAC Components

The subsequent sections dive deeper into each of these stages, clearly highlighting the specific role of core architectural components: Proxy Applications, TON Adapter, Sequencer Network, Executor mechanism, and TAC EVM Layer.

1. SDK

The TAC SDK makes it possible to create hybrid dApps that let TON users interact directly with EVM smart contracts without needing to manage multiple wallets or understand the complexities of cross-chain messaging.

The TAC SDK enables you to create frontends that:

- Connect to TON wallets like Tonkeeper or Tonhub
- Send transactions from TON to your EVM contracts

- Track cross-chain transaction status in real-time
- Handle tokens across both chains
- Create a seamless user experience for TON users

2. Cross-Chain Messaging

Cross-chain communication in TAC is centered around a deterministic and auditable message format. Each transaction initiated on the TON side is converted into a standardized cross-chain message that serves as the atomic unit of execution across chains.

These messages are constructed and emitted by the TON Adapter and are consumed by sequencers for indexing, validation, and Merkle tree construction. Once consensus is reached, they are executed on the TAC EVM side by a designated executor.

Message Format

Each message follows a structured schema to ensure interoperability, versioning, and security:

```
Message {
  uint32 timestamp;        // Transaction timestamp from the TON blockchain
  address target;          // Target smart contract address on TAC EVM
  string methodName;       // Method to call on the target contract
  bytes arguments;         // Method arguments (ABI-encoded)
  string caller;           // Original caller address on TON
  TokenAmount mint;        // Amount of tokens locked on TON
  TokenAmount unlock;      // Amount of tokens burned on TON
}
```

TAC ensures deterministic execution by enforcing a canonical, hashable message format suitable for Merkle proofs.

Version control across message fields guarantees future upgrades without breaking compatibility. Reentrancy protection is achieved through unique timestamps and identifiers embedded in each message. Auditability is maintained, allowing messages to be cryptographically verified both on-chain and off-chain at any time.

3. Security, sequencing and consensus

When voting for a new root, each participant submits their calculated root to the smart contracts. Once consensus on the new root is reached, the contracts establish an epoch with the new root.

Basic structure of an epoch in the blockchain (BC):

- `epoch_start_time` – the start time of the epoch — the timestamp of the BC block when the epoch began
- `epoch_delay` – the delay interval between the start of the current epoch and the voting for the new root
- `merkle_root` – the Merkle root valid for the current epoch
- `next_voting_time` – the time when voting for the new root begins

During the interval from `epoch_start_time` to `voting_time = epoch_start_time + epoch_delay`, the current epoch cannot be changed at the BC level, ensuring a guaranteed time for execution. This prevents a situation where some sequencers may operate slower than others and submit their votes right before the epoch ends, leaving no time for execution.

System Components
The system architecture consists of the following key components:

Sequencers – nodes responsible for collecting events from blockchains, forming cross-chain operations and Merkle roots, and achieving consensus when setting new roots.

Smart contracts in the EVM blockchain:

- **CrossChainLayer** – the main contract responsible for cross-chain operations
- **Consensus** – a contract implementing the consensus mechanism for establishing Merkle roots
- **Settings** – a contract containing protocol settings, including consensus thresholds and the maximum epoch duration

Smart contracts in the TON blockchain:

- **CrossChainLayer** – the main contract for cross-chain operations, storing Merkle roots
- **MultisigV1** – a multisignature contract used to reach consensus. Stores consensus thresholds

- **Settings** – a contract containing protocol settings, including the maximum epoch duration

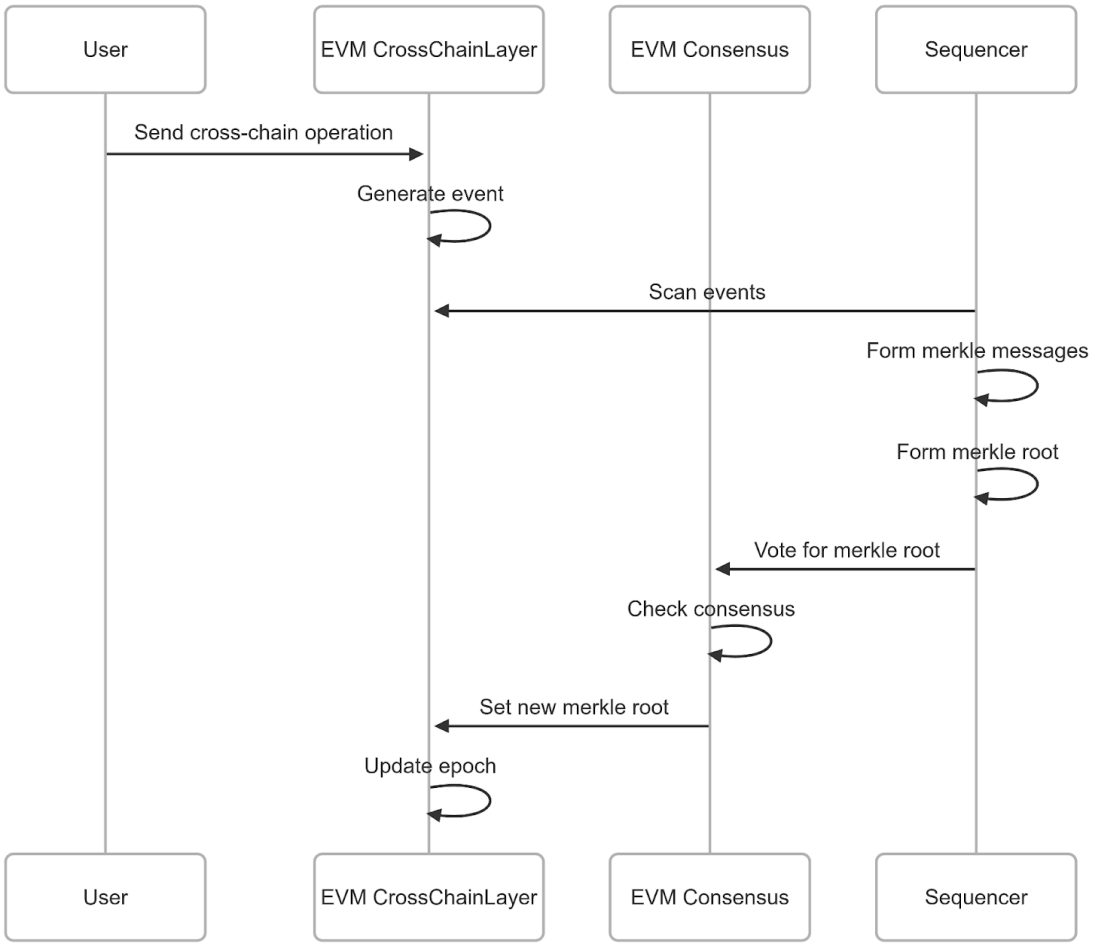
Component Interaction

Communication between the system components is organized as follows:

- **User → Smart contracts**: Users send cross-chain operations through smart contracts in any supported blockchain.
- **Smart contracts → Sequencers**: Smart contracts generate events (logs), which sequencers track and collect.
- **Sequencer → Sequencer**: Sequencers do not communicate directly with each other. Instead, they achieve consensus by voting within smart contracts.
- **Sequencers → Smart contracts**: Sequencers submit their votes for new Merkle roots to the consensus smart contracts.

Interaction Diagrams

Interaction for EVM:



Interaction for TVM:

Consensus in EVM

Consensus Contract

Consensus in the EVM blockchain is implemented via the `Consensus` smart contract. Key fields of the contract include:

- `_currentRoot`: the current Merkle root
- `_roots`: a mapping of all approved Merkle roots with their validity periods
- `_prevEpochStartTime`, `_currEpochStartTime`: start times of the previous and current epochs
- `_prevMessageCollectEndTime`, `_messageCollectEndTime`: end times of message collection for the previous and current epochs
- `_nextVotingTime`: the time when the next voting process can begin

Epoch Management:

Each epoch includes the following time markers:

- Epoch start time
- End of message collection
- Start of the next voting round

A new epoch begins when consensus is reached or can be forcibly initiated by the protocol administrator in case of consensus failure.
At the start of a new epoch, the set of voters may also be updated.

Voting Process

A sequencer calls the function `vote(messageCollectEndTime, newMerkleRoot)` in the `Consensus` contract — the contract verifies whether the caller is an authorized voter.
The contract checks that voting is active (i.e., the required delay since the previous epoch has passed).
The vote is recorded.
If consensus is reached, the new Merkle root is set and a new epoch begins.

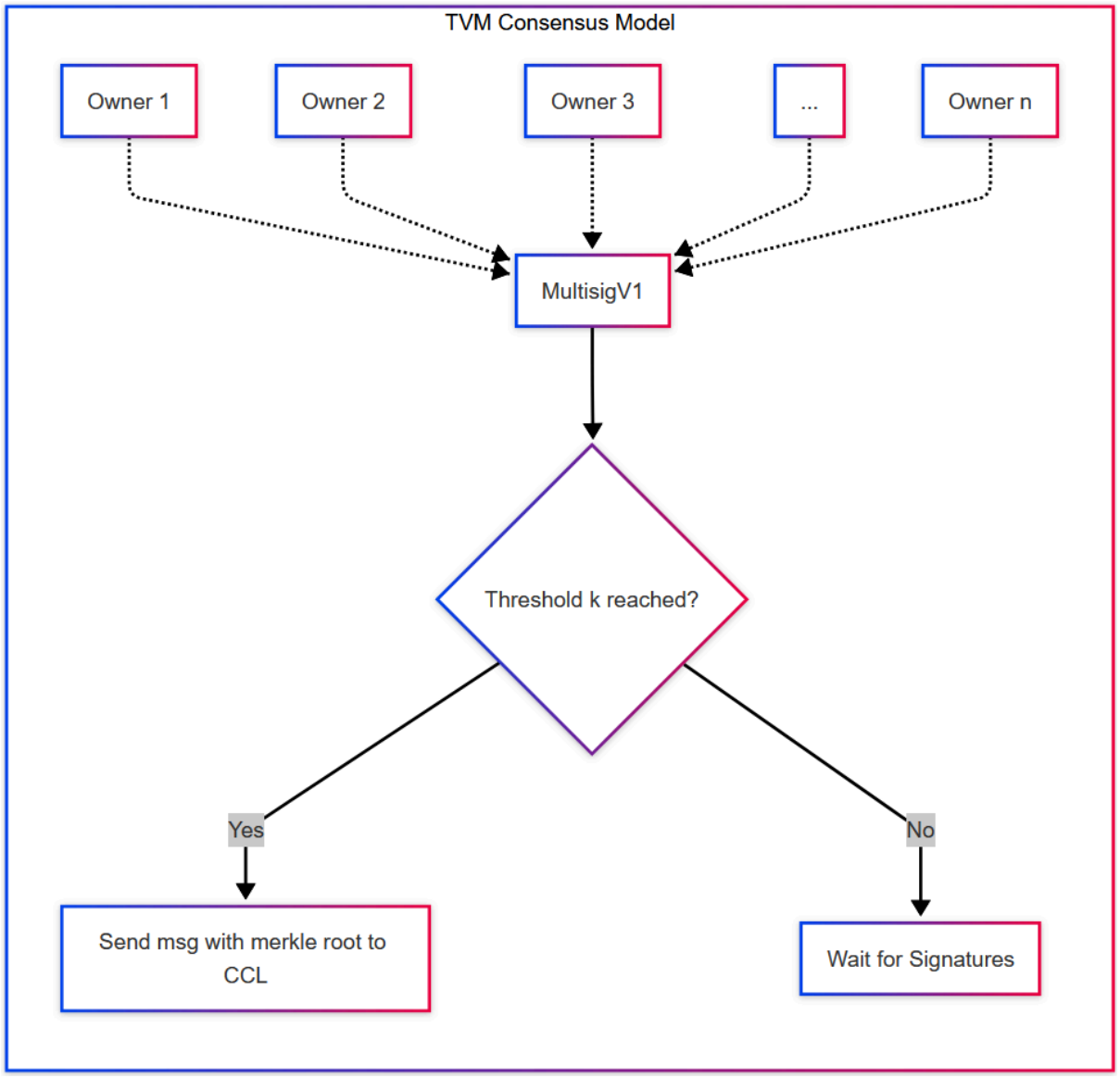
Voting message format:

- `messageCollectEndTime`: the end time of message collection for the current voting round
- `newValue`: the new Merkle root (`bytes32`)

Timing parameters:

- `EPOCH_DELAY_KEY`: delay between epochs (currently 10 seconds)
- `MAX_EPOCH_DURATION_KEY`: maximum duration of an epoch — after this period, an admin can update the set of voters and advance the epoch (currently set to 1 day in testnet)
- `VALIDITY_DURATION_KEY`: the validity period of a Merkle root

Consensus in TVM
Multisignature Contract



Consensus in the TVM blockchain is implemented through the `MultisigV1` contract, which stores public keys of signers at predefined indexes. It uses a multisignature mechanism that requires approval from a certain number of owners (k of n) to execute transactions.

Key fields of the contract:

- `wallet_id`: the wallet identifier
- `n`: total number of owners
- `k`: required number of signatures
- `owner_infos`: a dictionary containing information about owners — public keys mapped to predefined indexes
- `pending_queries`: a dictionary of pending requests — each entry stores data about the owners who signed the request. The key is `query_id`

Query structure:

- Query status (executed / not executed)
- Index of the query creator
- Number of collected signatures
- Bitmask of signing owners
- Message body

Signature Mechanism and Consensus Achievement

The signing and verification process in `MultisigV1` includes the following steps:

Request Creation:

- An owner creates a message containing the Merkle root update operation.
- The message is signed using the owner's key.
- The request is sent to the contract as an external message.

Signature Verification:

- The contract verifies the sender's signature using the `check_signature` function.
- It checks that the sender is an authorized owner (their public key is in `owner_infos`).
- It ensures the request is not expired and that the owner's spam limit has not been exceeded.

Signature Collection:

- If the request is new, it is saved in `pending_queries` with one signature.
- If the request already exists, the new signature is added.
- The signature count and the bitmask of signing owners are updated.

Transaction Execution:

- Once the number of collected signatures reaches the threshold `k`, the request is executed.
- A message is sent to the `CrossChainLayer` contract requesting to set the specified Merkle root.
- The request is marked as executed.

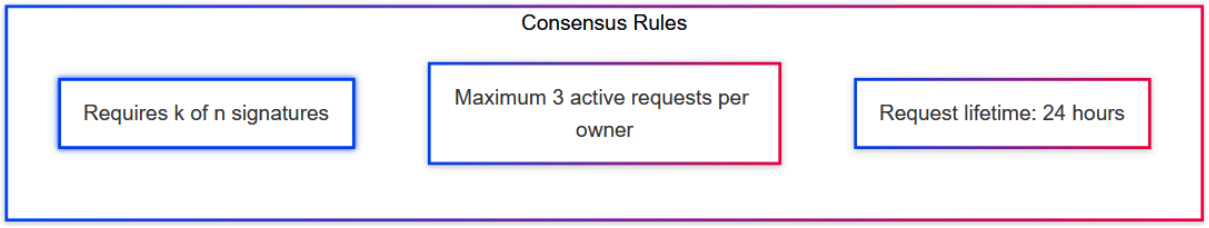
Message format for Merkle root update:

- External message with the signature of the owner sending it

- Dictionary of additional signatures *
- Wallet identifier
- Request identifier (`query_id`)
- Message body containing:
    - Address of the `CrossChainLayer` contract
    - Operation code `sequencerMultisig_updateMerkleRoot`
    - The new Merkle root
    - The end time of message collection for this root

* When sending a dictionary of additional signatures, the contract immediately verifies all included signatures.
It is therefore possible to send all `k` signatures within a single external message.
Note: TVM has specific logic for verifying multiple signatures — after 10 signature verifications, the gas cost for each subsequent verification increases 10-fold.

Attack Protection and Limitations



The `MultisigV1` contract includes several protection mechanisms:

`MAX_FLOOD`:

- Limits the number of active requests from a single owner (currently set to a maximum of 3 requests)
- Prevents DoS attacks where a malicious actor attempts to flood the contract with numerous requests
- When a request is either executed or expires, the flood counter is decreased
- If an owner reaches the `MAX_FLOOD` limit, they cannot create new requests until some of their existing requests are either executed or expired
    - However, they are still allowed to sign existing requests initiated by other owners

`QUERY_ID`:

- A unique request identifier — represents the expiration timestamp of the request

- Messages with the same `query_id` will not be executed twice — this prevents replay attacks
- The contract enforces a **minimum** `query_id`: it must not be less than the current blockchain time
- The contract also enforces a **maximum** `query_id`: requests with `query_id` older than 24 hours are rejected

`cleanup_queries` Function:

- Removes requests that expired more than 60 seconds ago (based on blockchain time)
- Cleans up old entries from `pending_queries`
- Decreases the flood counters for owners whose requests have expired
- Can be called by any user to clean up the contract

Timing Parameters:

- `EPOCH_DELAY`: the delay between epochs (currently 30 seconds in the testnet)
- `MAX_EPOCH_DURATION`: the maximum duration of an epoch — does not affect root setting or consensus in TVM, used only for reference by sequencers (currently 1 day in testnet)

Root Submission and Sequencer Synchronization

Merkle Root Formation and Submission

The basic Merkle tree formation process in the sequencer follows these steps:

1. Epoch Retrieval from Contracts
The sequencer retrieves the current epoch data from the smart contracts.

2. Blockchain Synchronization Check
The sequencer checks whether it is synchronized with the current state of the blockchain:

- It verifies that all events up to a certain time have been processed.
- If the sequencer is not synchronized, the root submission process is delayed — a recovery mode is triggered.
- In recovery mode, the sequencer re-collects events starting from the last synchronization point.
- Tree formation will resume only after full synchronization. Once the generated root matches others, recovery mode is turned off.

3. Voting Activity Check

- The sequencer verifies whether it is time to vote.
- It ensures that it has collected all events up to the `NextVotingTime` of the current epoch.

4. Tree Construction:

- A time interval is defined for collecting messages: `NextVotingTime` of the current epoch + `n * delay` (initially, `n = 1`).
- It confirms that all events in this interval have been processed.
- Messages are organized into Merkle tree layers. At each level, hashes of pairs of lower-level nodes are computed. The process continues until a single root hash (Merkle root) is obtained.

5. Validity Check:

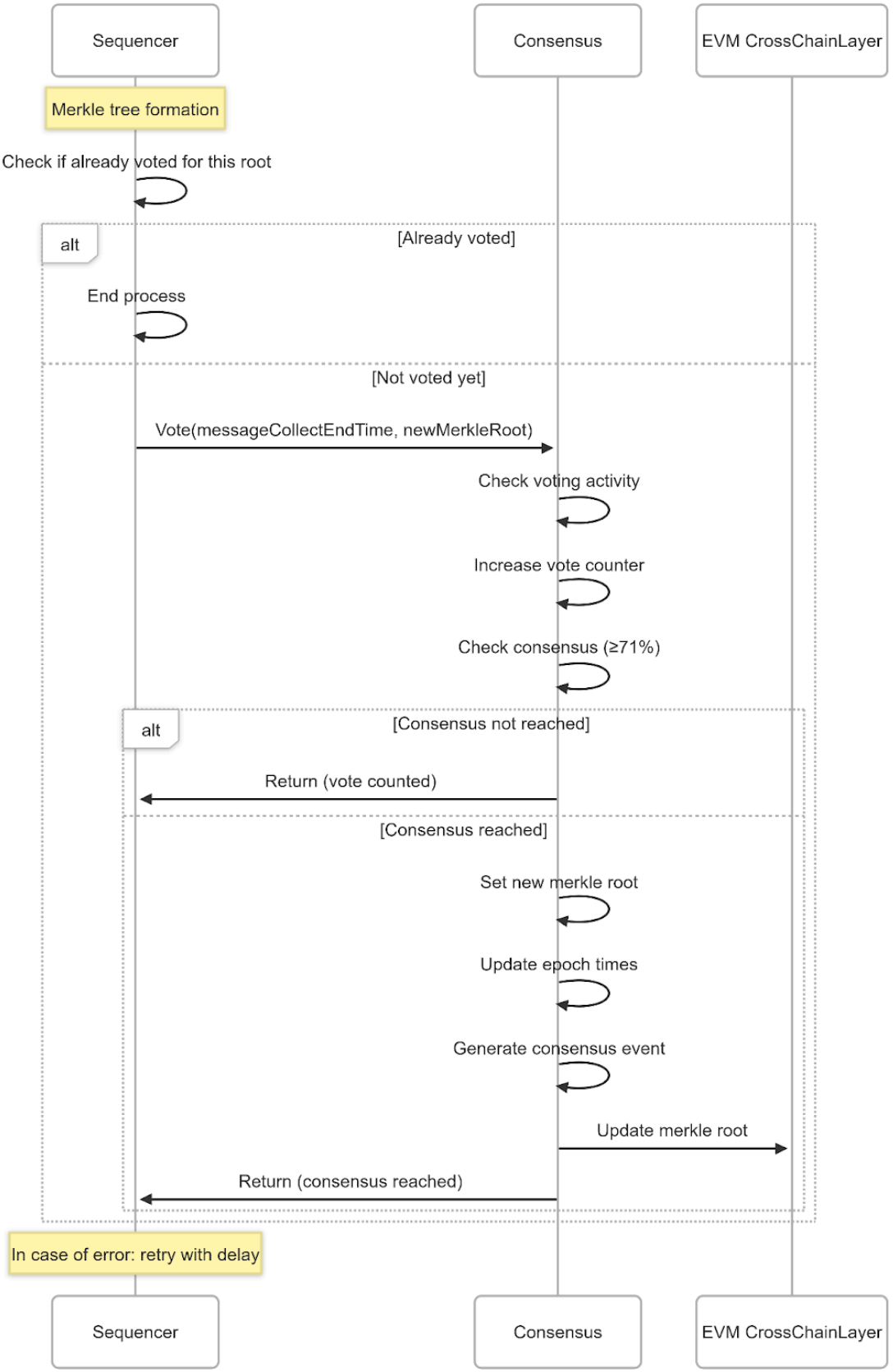
- The new Merkle root is checked against the current root to ensure it has changed.
- It is also verified that the new root is non-zero (i.e., there are messages).
- If the root has not changed and the maximum epoch duration (`MAX_EPOCH_DURATION`) has not been reached, the collection interval is increased (`n + 1`) and tree construction is restarted.
- If no valid root can be built, the process is postponed.

6. Root Submission to Contracts

- A message with the new Merkle root is prepared.
- The message is sent to the appropriate blockchain contract.
- If an error occurs during transaction submission, the process is aborted and will restart during the next cycle.

7. Tree Persistence Upon Successful Submission:
The Merkle tree is saved in the database with the following data:

- Root hash
- Message collection end time (`MessageCollectEndTime`)
- Tree layers for proof generation — a dictionary of messages (`Dict`) for TVM
- Creation time and blockchain status (initially marked as "not submitted")



Merkle Root Submission in EVM

Process of submitting a Merkle root in the EVM blockchain:

1. Voting Preparation:

- The sequencer checks whether it has already voted for the given Merkle root.
- If it has already voted, the process completes successfully.
- If not, a voting transaction is prepared.

2. Vote Submission:

- The sequencer calls the `vote` method of the `Consensus` contract, passing:
  - The message collection end time
  - The new Merkle root (as `bytes32`)
- The transaction is sent to the network.

3. Result Handling:

- The sequencer waits for the transaction to complete and checks its status.
- In case of failure, retries are performed with a delay.
- The maximum number of attempts is limited by the sequencer's configuration.

4. Consensus Achievement in the Contract:

- The `Consensus` contract processes the vote and updates the vote counter.
- It checks whether the consensus threshold is reached.
- Once consensus is reached:
  - The new Merkle root is set.
  - A consensus event is emitted.
  - A new epoch starts with updated timestamps.

Merkle Root Submission in TON

Process of submitting a Merkle root in the TON blockchain:

1. Message Preparation:

- The sequencer creates a message with the Merkle root update operation.
- The message includes:
  - The new Merkle root (256 bits)
  - The message collection end time (48 bits)
  - The time-quantized `queryId` (64 bits)

2. Multisig Contract State Check:

- The sequencer retrieves data from the `MultisigV1` contract.
- It verifies the signer's index (`SignerIndex`).

- It checks that the flood limit (`MAX_FLOOD = 3`) has not been exceeded for this signer.
- If needed, expired requests are cleaned up to reduce flood count.

3. Balance Check:

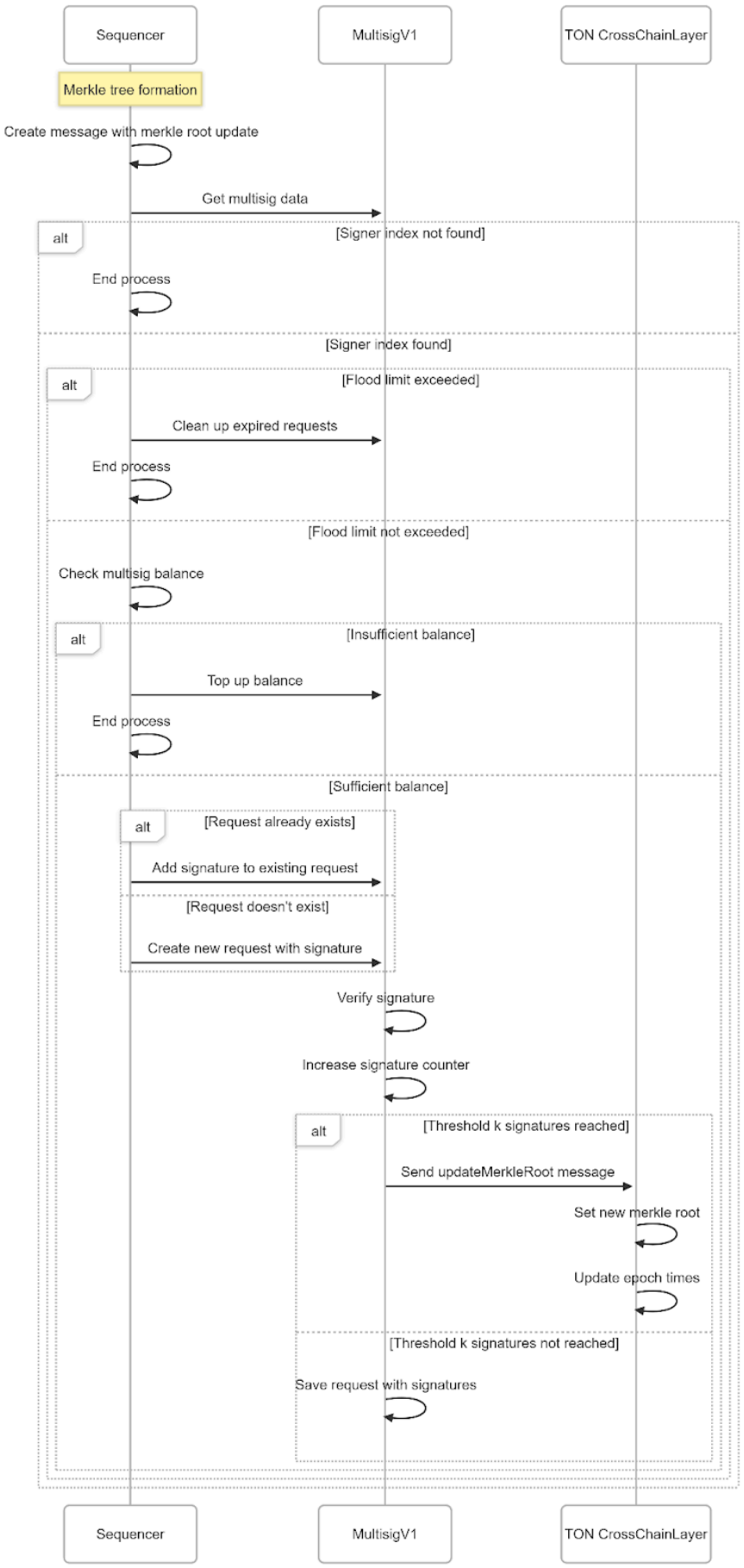
- The sequencer ensures that the `MultisigV1` contract has sufficient balance.
- If not, it initiates a top-up from the sequencer's wallet — the operation is deferred to the next cycle.

4. Signature Submission:

- If the request already exists, the sequencer adds its signature to the existing request.
- If not, a new request is created with the sequencer's signature.
- The message is signed with the sequencer's key and sent to the `MultisigV1` contract.

5. Processing in `MultisigV1` Contract:

- The contract verifies the sender's signature.
- The signature is added to the request, increasing the signature count.
- If the threshold `k` is reached:
    - The request is executed.
    - A message is sent to the `CrossChainLayer` contract.
    - `CrossChainLayer` sets the new Merkle root.
    - The epoch is updated.


Sequencer
MultisigV1
TON CrossChainLayer
Merkle tree formation
Create message with merkle root update
Get multisig data
alt
[Signer index not found]
End process
[Signer index found]
alt
[Flood limit exceeded]
Clean up expired requests
End process
[Flood limit not exceeded]
Check multisig balance
alt
[Insufficient balance]
Top up balance
End process
[Sufficient balance]
alt
[Request already exists]
Add signature to existing request
[Request doesn't exist]
Create new request with signature
Verify signature
Increase signature counter
alt
[Threshold k signatures reached]
Send updateMerkleRoot message
Set new merkle root
Update epoch times
[Threshold k signatures not reached]
Save request with signatures
Sequencer
MultisigV1
TON CrossChainLayer

Synchronization

Synchronization serves the following purposes:

- Ensures data consistency between sequencers and blockchains
- Prevents conflicts during Merkle root submission
- Enables recovery from failures and data discrepancies
- Coordinates sequencer actions without direct communication between them

The synchronization process includes the following steps:

1. Blockchain Event Scanning:

- The sequencer scans events in the blockchains (EVM and TVM)
- Events are sorted by time and type
- Special attention is given to Merkle root submission events

2. Consistency Check:

- The sequencer compares its local state with the blockchain state
- Merkle roots stored in the database are compared with those in epochs
- If a discrepancy is detected, recovery mode is triggered

3. Pre-Vote Synchronization:

- Before forming a new Merkle root, the sequencer verifies synchronization with the blockchains
- It ensures that the message collection end time does not exceed the latest scan timestamp of the other blockchain
- If the sequencer is not synchronized, voting is postponed

4. Handling Merkle Root Submission Events:

- Upon detecting a new Merkle root submission event in the blockchain:
    - Information about the established epoch is extracted
    - If the root from the local tree matches the one from the event, it is marked as synchronized.
    - From this point forward, the sequencer will only look for inconsistencies beyond this tree.

Recovery Mode

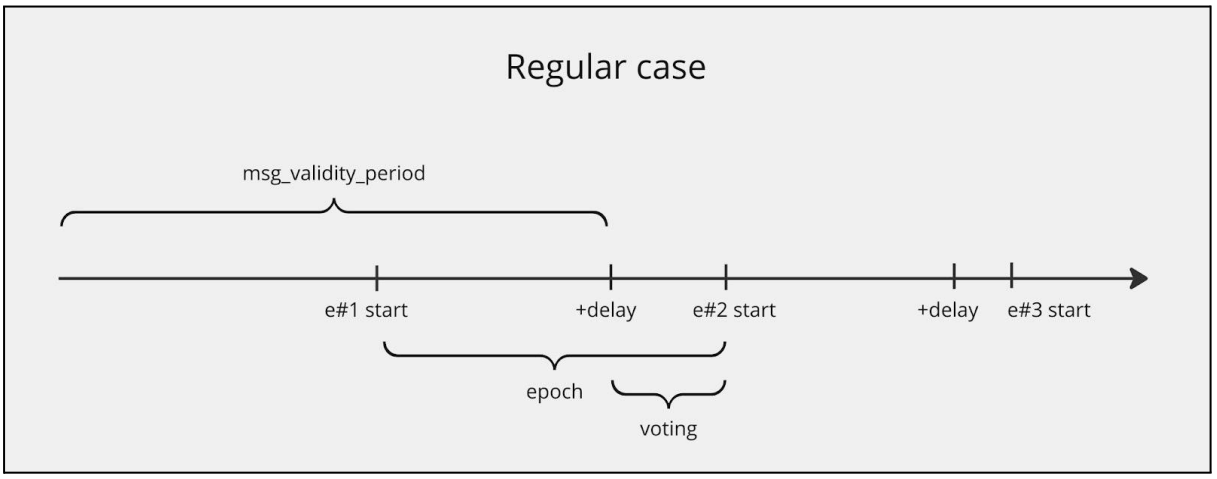
Recovery mode is activated in the following scenarios:

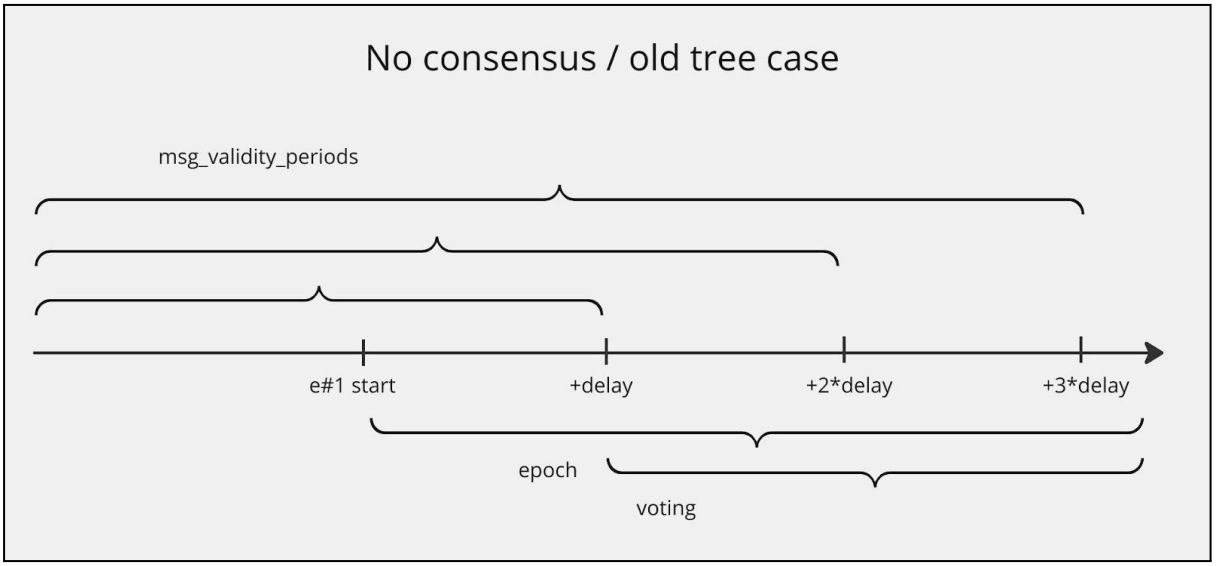
- **Merkle root discrepancy:** When the locally constructed Merkle root differs from the one set in the blockchain, which can happen due to missed events
- **No consensus is reached** within the maximum epoch duration

In recovery mode:

- The sequencer pauses the formation of new Merkle roots
- A re-scan of blockchain events is performed starting from a specific time — either after the last synchronized tree or covering the entire message validity period for the current epoch
- The local state is synchronized with the blockchain state
- Once synchronization is achieved, recovery mode is deactivated

Epoch Parameters (epochDuration, epochDelay)





TAC uses a strictly defined epoch-based system to synchronize message collection and consensus across the entire network:

- **epochDuration:**
  Defines the exact length of an epoch. Sequencers gather and batch messages during this predefined period.
- **epochDelay:**
  Establishes a fixed delay interval during which no changes to the epoch state (such as incrementing or restarting voting) are permitted. This guarantees consistent and deterministic tree submissions, even under varying network conditions.

These epoch parameters provide clear timing rules for sequencers, ensuring deterministic synchronization and preventing timing-based consensus attacks.

Root Finalization Process (Tree → Vote → Submit)

The finalization of each epoch's Merkle root follows a sequential and transparent procedure:

1. **Tree Assembly:**
   Individual sequencers independently construct their Merkle trees based on locally collected cross-chain messages.
2. **Network Submission:**
   Each sequencer signs the received Merkle root and submits the signature to the smart contract. Once consensus reaches 3 out of 5 signatures, it is formally finalized.
3. **Execution Trigger:**
   After the Merkle root is set, executors process the messages assigned to them on the TAC EVM Layer.

Binding to Merkle Roots and Epochs

Message uniqueness and reentrancy protection are tightly coupled to Merkle roots and epoch definitions:

- Each finalized Merkle root cryptographically anchors message batches to a specific epoch, creating immutable references.
- Messages from past epochs cannot be reexecuted or modified, since epoch transitions provide a deterministic cut-off and strong temporal boundary for message validity.

4. EVM-side Proxy Contract

EVM-side Proxy Contracts serve as the execution entry points for Hybrid dApps on TAC. Developed by a dApp and deployed alongside target contracts, they decode cross-chain messages, validate Merkle proofs, and execute the corresponding calls on the TAC EVM Layer.

If required, Cross-chain layer smart contract handles asset minting or unlocking on the Proxy before invocation, and can emit return messages back to TON. This modular structure allows dApps to support cross-chain flows without altering their core Solidity logic.

5. Executing

A separation of responsibilities between cross-chain message sequencing and execution.
This division simplifies the implementation of slashing mechanisms and the selection process for message executors.

In future versions of the cross-chain protocol, the **Executor** will provide oracle data during message execution, enabling access to the most up-to-date data.

Executor

Responsible for collecting cross-chain messages and executing them on both TON and TAC networks.
Each message specifies the executor eligible to perform it, and this executor receives the associated fee.

Executor Details

Message execution requires submission of its parameters and tree hashes, ultimately forming a Merkle root.
Security is guaranteed through cryptographic proofs, so no collateral or centralized approval is required for executors.

Each cross-chain message must specify which executor is authorized to process it. The cross-chain smart contract will verify that the message is executed by the designated executor.

For messages traveling the path **TON → TAC → TON**, it is necessary to specify the addresses of **two executors**, one for each execution step.

Transaction Execution on TAC EVM

Execution involves submitting the finalized message batch along with Merkle proofs to the **cross-chain layer smart contract** deployed on the TAC EVM Layer. The process includes:

- **Proof Verification:**
  The cross-chain layer smart contract checks that submitted messages match the finalized Merkle root, ensuring that only fully consensus-approved messages are executed.
- **Asset Handling:**
  Any assets previously locked on the TON blockchain are represented as minted (wrapped) tokens on TAC EVM. Conversely, tokens burned on TON can be unlocked and made available for use on the TAC EVM side. This lock-mint and burn-unlock mechanism maintains token supply integrity across chains.
- **Smart Contract Execution:**
  Once asset states are synchronized, the specified method of the target smart contract (identified by the "target" address and method arguments from the cross-chain message) is executed on TAC's EVM environment.
- **Result Propagation:**
  Execution results (success or failure, token balances, updated states) are immediately reflected on-chain.

Rollback Mechanism

TAC also incorporates robust rollback logic to handle any failed transactions:

- **Detection and Identification:**
  If a transaction fails execution on TAC's EVM, sequencers add this message to a separate Merkle rollback tree.
- **Rollback Execution:**
  Once approved through a similar consensus process, locked tokens are automatically and securely returned to the original sender on the TON blockchain.
- **Automatic Handling:**
  Rollbacks require no manual intervention (if enough commission) from users or developers, providing additional security and convenience.

Transaction Atomicity and Rollbacks

TAC ensures transaction atomicity through comprehensive rollback mechanisms. If a transaction fails at any point during cross-chain execution, the system guarantees the safe return of assets to the originating chain.

Rollback Merkle Tree Creation

When execution failures occur on the TAC EVM Layer, the failed messages are immediately recorded and grouped into a special rollback Merkle tree:

- This rollback tree has the same deterministic properties as regular transaction trees.
- It clearly identifies failed messages, simplifying network-wide agreement on rollback operations.

Consensus on Rollbacks

Rollback trees are submitted through a similar sequencer consensus process used for regular transactions:

- Sequencers reach consensus on rollback Merkle roots, ensuring transparent and secure approval.
- Once finalized, these rollback operations become authoritative and immutable.

Safe Return of Locked Assets on TON

Upon finalization of rollback consensus, locked assets on the TON blockchain are automatically and securely released back to users:

- No manual intervention(if enough commission) is required from users or developers.
- Assets remain locked only if execution fully succeeds, preventing loss or lock-up due to transaction failures.

Rollbacks as a Core UX Guarantee

Automated rollback logic is fundamental to TAC's user experience:

- Users always have clear guarantees: their assets are either successfully transferred or returned.
- This mechanism significantly reduces user risk and enhances confidence in cross-chain interactions.

6. Fee Settlement

Finally, transaction execution concludes with the settlement of transaction fees. TAC employs a directional fee model, simplifying the user experience:

- Fees for operations originating on TON are paid exclusively in TON tokens; operations originating on TAC pay fees exclusively in TAC tokens.
- Fees include two components: a fixed **Sequencer fee** (distributed among active sequencers based on collateral stake) and an **Executor fee** (covering gas costs and execution overhead on the EVM side).

By the time execution completes, all stakeholders receive compensation, closing the transaction loop and fully reconciling state across both the TON and TAC EVM chains.

The described process—finalized consensus, deterministic execution, rollback guarantees, and transparent fee settlements—ensures TAC provides reliable, secure, and frictionless integration of Hybrid dApps between TON and EVM ecosystems.

Fee and Incentive Model

TAC implements a directional, EIP-1559-compatible fee system optimized for cross-chain execution between TON and the TAC EVM Layer. The model is designed to balance user simplicity with accurate protocol-level accounting, while ensuring proper compensation for sequencers and executors.

Directional Fee Logic

The fee token is determined by the origin of the transaction:

- **TON → TAC (→ TON) - most :** all fees are paid in TON.
- **TAC → TON:** fees are paid in TAC.

This model ensures single-token UX for users, allowing transactions without holding TAC directly. A network-level paymaster contract converts TON to TAC on behalf of users when needed.

Fee Structure

Each cross-chain message includes:

- **Sequencer fee** – a fixed amount, set in protocol parameters, distributed among sequencers proportionally to their stake [and sequenced messages done];
- **Executor fee** – the remaining amount of totalFee – sequencerFee, used to cover gas costs on the destination chain and compensate the executor.

For round-trip transactions (e.g., TON → TAC → TON), the message includes:

- Two executor references;
- Two executor fee allocations (pre-reserved from the TON side);
- A flag marking the return message as prepaid.

Gas Pricing Model (EIP-1559 Compatible)

The TAC EVM Layer follows Ethereum's EIP-1559 model:

- baseFee is burned;
- priorityFee (tip) is paid to the block proposer;
- maxFeePerGas is the upper bound for total fee per unit gas.

The actual priority fee is adjusted as:

priorityFee = min(maxPriorityFeePerGas, maxFeePerGas – baseFee)

Block proposers prioritize transactions by priorityFee × gasUsed to maximize income.

User-Side Fee Estimation

Before sending a message, the user performs:

1. Call to simulateReceiveMessage() to estimate gas usage;
2. Retrieval of:
    i. Current baseFeePerGas;
    ii. Suggested maxPriorityFeePerGas;
3. Calculation of:

maxFeePerGas = baseFee × 2 + maxPriorityFee

totalFee = estimatedGas × maxFeePerGas

3. Conversion of totalFee to TON or TAC using an oracle-based USD exchange rate;
4. Attachment of the final amount to the message payload.

Executor-Side Fee Verification

Before executing a message, the executor:

- Estimates gas usage using the Merkle proof;
- Reads current baseFee and priorityFee;
- Recalculates maxFeePerGas using the same formula;
- Compares calculated fee with the attached amount.

If the attached fee is insufficient, execution is postponed. Executors may subsidize small deficits to preserve UX, at their discretion.

Fee Fixation

- **Sequencer fee** is fixed at submission time on the source chain;
- **Executor fee** is finalized at the time of execution on the destination chain;
- This hybrid model reduces UX complexity while maintaining execution integrity.

Risk Mitigation

To address volatility and execution failure:

- The protocol recommends setting maxFeePerGas to approximately 2 × baseFee;
- Executors may delay underfunded transactions but are permitted to process them if viable;
- Prepaid return messages ensure completion without requiring additional gas deposits.

This fee model provides consistent incentives for protocol participants while enabling user-friendly, asset-native interactions across TON and TAC environments.

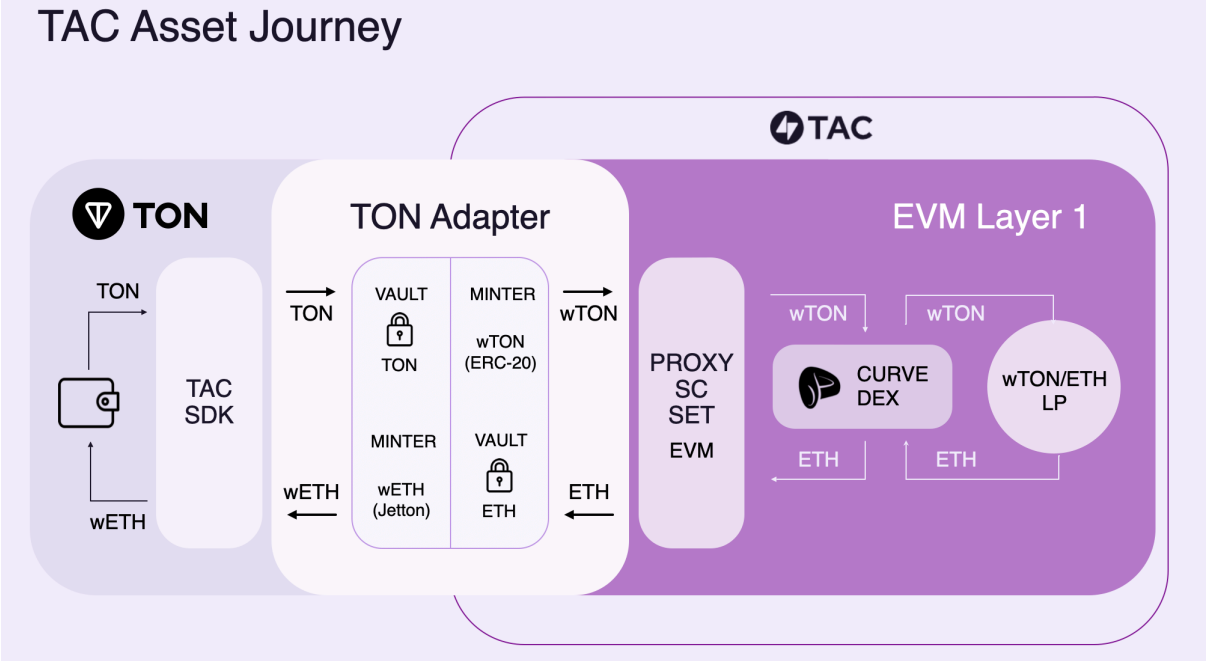
User Interaction Flow

From the user's perspective, TAC operates entirely within the TON environment. A transaction initiated through a TON wallet interacts with TAC SDK and sends tx to Cross-chain layer smart contract. This contract locks the relevant asset and emits a cross-chain message that is processed by sequencer nodes and executed on the TAC EVM chain. The result is returned to the user's wallet. User doesnt need to wrap assets, use third-party bridges, or external wallet interfaces.

This model is designed to minimize cognitive overhead for users while maintaining protocol-level guarantees for correctness, reentrancy protection, and asset safety.

The following steps illustrate a simplified end-to-end journey of a transaction through TAC:

1. **User Transaction Initiation**
   A user initiates a transaction from their TON wallet to interact with an EVM dApp via a TAC SDK.
2. **TON Adapter Processing**
   The TON Adapter locks assets and emits a standardized cross-chain message containing execution instructions and asset details.
3. **Sequencer Indexing & Batching**
   Off-chain sequencers monitor and index these messages, periodically aggregating them into Merkle trees for consensus.
4. **Consensus Across Sequencers**
   Sequencers reach consensus (network-level) to validate and finalize the transaction batch.
5. **Executor Selection & Transaction Execution**
   The assigned executor submits the Merkle hashes to the TAC EVM Layer and triggers the corresponding smart contract logic via Proxy Apps.
6. **Result Handling and Rollback Mechanism**
   TAC EVM executes the transaction. Upon success, assets and state are updated accordingly. If execution fails, an automated rollback mechanism safely returns locked assets to the original sender.
7. **Fee Settlement**
   Fees are processed using TAC's directional fee model, ensuring that users pay fees in their native chain token (TON or TAC), automatically distributed among sequencers and executors according to protocol rules.



Lifecycle of a Message

Once created by the TON Adapter, a message passes through the following lifecycle on its way to execution:

1. **Emission**
   The message is emitted by the TON Adapter smart contract after validating transaction structure and asset locks.
2. **Indexing by Sequencers**
   Off-chain sequencers detect and locally store the message within the current epoch window, tagging it with its timestamp and metadata.
3. **Merkle Tree Assignment**
   At the end of the epoch, sequencers collected messages into a Merkle tree. Each message is encoded deterministically, allowing for uniform tree construction across nodes.
4. **Merkle Root Generation**
   The Merkle root generated from the message batch becomes the cryptographic commitment used in TAC's cross-chain consensus process.
5. **Execution Trigger**
   Once a message is included in a finalized Merkle root and consensus is reached, it can be executed on the TAC EVM Layer by a designated executor, ensuring atomic and verifiable state transitions.

Reentrancy Protection and Message Uniqueness

To ensure message integrity and prevent reentrancy attacks, TAC incorporates robust protection mechanisms at multiple protocol layers, from message generation through final execution.

Unique Message IDs and Timestamps

Every cross-chain message generated by the TON Adapter is uniquely identified through:

- **Timestamping:**
  Each message includes a precise timestamp recorded at the time of transaction initiation on the TON blockchain, preventing reentrancy at different epochs or timeframes.
- **Unique Identifiers (IDs):**
  Transaction hashes and Merkle proofs uniquely tie every message to a specific state, ensuring that each message is distinct and easily verifiable.

This uniqueness ensures that every transaction execution is singular, deterministic, and securely traceable.

Reentrancy Checks at Adapter and Executor Levels

Reentrancy protections are enforced at two critical points within the TAC infrastructure:

- **Adapter-Level Verification:**
  On the TON side, the Adapter contract verifies incoming transactions, checking timestamps and unique message IDs to prevent previously executed or duplicate transactions from re-entering the processing pipeline.
- **Executor-Level Verification:**
  Prior to execution on TAC EVM, the executor confirms via the cross-chain layer contract that each message has not already been executed or invalidated.

Single-Execution Guarantees

All TAC cross-chain messages are processed exactly once or not at all. This atomicity is critical for maintaining consistent state across both the TON and TAC EVM chains:

- If a message execution is successful, the corresponding state changes (mint/unlock assets, contract calls, etc.) occur precisely one time.
- Failed executions trigger automatic rollbacks, ensuring that no intermediate or inconsistent state can persist.

Economic Security and Incentives

To ensure reliable participation and alignment of incentives, TAC employs a comprehensive economic security model across sequencers.

Collateral Requirements (stTON, stETH, etc.)

Each sequencer is required to maintain collateral above a DAO-defined minimum. Acceptable collateral includes:

- **TON Liquid Staking Tokens (stTON)**
- **Wrapped staked ETH (wstETH, stETH)** (planned to have at day 1, but subject to Agglayer going live)
- Other DAO-approved liquid staking derivatives

This collateral acts as an economic guarantee of honest behavior, ensuring participants have a significant stake at risk in case of malicious or negligent actions.

Slashing for Malicious Behavior

The protocol employs slashing penalties to economically disincentivize dishonest or malicious behaviors:

- **Invalid or Conflicting Root Submission:**
  If a sequencer submits a false or conflicting Merkle root, any other sequencer can initiate an on-chain penalty claim. Upon verification, part of the offending sequencer collateral is slashed.
- **Preventative Economic Security:**
  Credible slashing penalties ensure malicious actions are economically irrational, significantly raising attack costs.

Reward Distribution (Sequencers and Executors)

Economic incentives are provided through clear reward mechanics:

- **Sequencer Rewards:**
  Fixed sequencer fees (collected per transaction) are distributed among all sequencers proportionally based on collateral stakes.

- **Executor Rewards:**
  Executors are additionally compensated with dynamic executor fees, covering gas and execution overhead on the EVM Layer.

This economic structure balances predictable base income for honest participation with additional incentives for active executor roles.

Future Security Enhancements

TAC plans to continually strengthen its security model by integrating advanced cryptographic and economic security methods.

Babylon Bitcoin Restaking

Babylon Bitcoin restaking represents a planned security layer leveraging the economic strength of the Bitcoin blockchain:

- **BTC as an External Economic Security Layer:**
  Bitcoin holders delegate BTC stakes to Babylon validators, who then cryptographically verify TAC EVM blocks and Merkle roots.
- **Self-Custodial Delegation:**
  Bitcoin holders maintain custody of their BTC, ensuring security without centralization risk.
- **Slashing for Equivocation (Double-Signing):**
  Validators who provide conflicting signatures (equivocation) or malicious attestations are economically penalized directly via the Babylon protocol, significantly increasing the cost of potential attacks.

Planned FROST Consensus Upgrade

Decentralized Threshold Signatures FROST-Ed25519 + DKG
Introduction

The FROST protocol (Flexible Round-Optimized Schnorr Threshold) is implemented (following RFC 9591), which allows a group of participants to jointly generate Schnorr signatures. Our project presents FROST-Ed25519, an adapted version of FROST for creating threshold signatures in the Ed25519 format (RFC8032), compatible with standard verification (e.g., in TVM).

Distributed Key Generation (DKG)

The initial process of generating the shared public key A and distributing secret key

shares s_i of the secret s is critically important for the security of the entire system. Our version of the DKG protocol is designed with the following principles:

- **Dealerless:** Participants jointly generate the key and its shares. No one ever knows the entire secret s.
- **High threshold:** We support a DKG configuration where more than half of the participants ($k > n/2$) are required to reconstruct the secret s.
- **Feldman VSS:** Feldman's Verifiable Secret Sharing scheme is used as the basis, allowing participants to publish "commitments" to their actions.
- **Confidentiality of shares with provable correctness:** Secret shares are transmitted between participants in encrypted form. Each participant, when sending an encrypted share, attaches a publicly verifiable cryptographic proof that the encrypted share is correct and corresponds to the previously published commitments. This proof does not reveal the share itself.
- **Complaint handling without secret disclosure:** If a participant receives an incorrect share (as revealed by verifying the attached proof), they may file a complaint. Any network member can verify this complaint using only publicly available data (ciphertext, proof, sender's commitments). The sender's fault is proven cryptographically without the need for anyone to disclose the actual secret share. This ensures that each complaint leads to unambiguous consequences either for the dishonest sender (disqualification) or to the rejection of an unfounded complaint.
- **Proof techniques used:** We apply Verifiable Encryption (PVE), built on non-interactive zero-knowledge proofs (NIZK) based on Sigma protocols (e.g., Chaum-Pedersen) and the Fiat-Shamir heuristic. These methods do not require a trusted setup.
- **Consensus:** All public DKG data (commitments, proofs, complaints) are recorded and ordered through decentralized BFT consensus (e.g., Tendermint), operating under a weak synchrony model.

Key Features of FROST-Ed25519

- **Threshold Security:** Creating a signature requires participation from t+1 out of n participants. An attacker controlling up to t participants cannot forge a signature or compromise the shared secret key.
- **Ed25519 Compatibility:** Generated signatures are fully compatible with the Ed25519 standard and can be verified using existing libraries and systems.
- **Round Optimization:** FROST is designed to minimize the number of communication rounds during signing. In our implementation, we offer a mode without a dedicated aggregator.

- **Decentralization:** All stages, including key generation and signature aggregation, are performed in a decentralized manner by the participants.

Our Implementation of FROST-Ed25519: Key Enhancements

Our approach to implementing FROST-Ed25519 includes the following important modifications and emphases aimed at enhancing decentralization and flexibility:

- **Fully Decentralized Signature Aggregation:**
  In accordance with Section 7.5 of RFC 9591, we have eliminated the role of a central signature aggregator. Each participant involved in signing independently receives partial signatures and commitments from other participants, verifies their correctness, and locally computes the final threshold signature.
- **Flexible Generation of One-Time Values (Nonces):**
  Standard FROST assumes a preprocessing phase where participants generate multiple pairs of one-time secrets (d_i□, e_i□) and their corresponding public commitments (D_i□, E_i□) for future use. This allows the signing process to be reduced to two rounds. In our implementation, we support generating one-time pairs (d_i, e_i) and commitments (D_i, E_i) immediately before each signing act. This adds one communication round (total of 3 rounds per signature) but eliminates the need to store and manage a large number of pre-generated values.
- **Important Note:** Even with the 3-round approach, the first round (generation and exchange of commitments (D_i, E_i)) for different messages can still be conducted in advance and independently. If participants pre-generate a set of such commitments, then the actual process of signing a specific message can still appear as a two-step procedure.

Share Refreshing and Changing the Set of Sequencers

One of the key challenges of long-lived threshold cryptography systems is the need to adapt to changes: participants may leave or join the system, and security requirements (and therefore the threshold value) may evolve over time.
Our system addresses this challenge through several subprotocols that allow for:

- **Changing the Committee Composition:** Securely adding new participants and removing old ones.
- **Changing the Threshold Value:** Adapting the reconstruction threshold k (and the associated parameter t) to new security requirements or the size of the committee.

- **Refreshing Shares (Proactive Security):** Periodically generating new shares for the same secret, rendering old shares useless and protecting against gradual node compromise (“mobile adversary”).

How Resharing Works:
The resharing process is fully decentralized and initiated through the consensus of the current committee. The core idea is that each participant of the current committee (P_i with old share s_i) acts as a "mini-dealer" for their share.

- If only proactive share refreshing is needed, the constant term can be set to 0.
- If a new participant needs to be added without changing the threshold, a simplified variant is also possible (this specific detail may still be subject to change).

3. TAC Token and Tokenomics

Tokenomics

TAC is the native utility token of the TAC network. It plays a central role in execution, consensus, and governance processes. Implemented as an ERC-20 asset on the TAC EVM Layer, it is used across all layers of the protocol.

$TAC role inside the Protocol

- **Gas Token:** TAC is the gas token for the TAC EVM Layer. Every EVM transaction ultimately consumes TAC. For TON-originating transactions, the network-level paymaster automatically converts user-attached TON fees into TAC.
- **Staking and Security:** TAC is staked by validators securing the TAC EVM Layer. Staked TAC backs network security and is subject to slashing for malicious behavior.
- **Governance:** Staked TAC grants voting rights within the protocol's DAO, governing upgrades, inflation parameters, and system-wide configurations.

TAC’s fundamental value is reinforced by these pillars while offering a seamless TON-first user experience.

Supply and Emission

- **Initial Supply:** 10,000,000,000 TAC minted at Token Generation Event.

- **Inflation:** to fund validator rewards and motivate staking. MAx inflation is planned at 3-5% range with linear parametric function peaking at goal_bonded rate. In case the staking rate is lower or higher than goal_bonded, the inflation rate is lower then the maximum level
- **Vesting:** Allocations to the team, investors, and ecosystem incentives are distributed over multi-year linear schedules to ensure long-term alignment.

4. Governance and Upgradeability

TAC is governed by an on-chain DAO, which oversees protocol parameters, validator elections, sequencers, and upgrade processes. The DAO is central to ensuring long-term adaptability, security, and decentralization of the TAC network. TAC uses a standard CosmosSDK governance module

Governance Scope

The DAO manages key protocol-level mechanisms and parameters, including:

1. **Consensus thresholds** — such as sequencer consensus (3/5).
2. **Epoch system settings** — including epoch_duration, epoch_delay, max_epoch_duration, and msg_validity_period.
3. **Sequencer lifecycle** — rotation schedules, and disqualification rules.
4. **Collateral requirements** — minimum stake required to operate as a sequencer.
5. **Slashing conditions and penalties** — including the threshold (e.g., 66%) required for validating slashing claims.
6. **Election cycle duration** — defining how frequently sequencer elections occur (parameter N).
7. **Protocol upgrades** — including support for FROST signatures, zk-proof systems, and EVM-level improvements.
8. **Economic parameters** — such as staking rewards, validator emissions, and DAO-managed budgets.

Upgrade Path and Protocol Evolution

The DAO governs all protocol upgrades, including:

- Validator set updates.
- EVM-level improvements (FROST, account abstraction).
- Economic rule adjustments (staking incentives, inflation curves).

- Technical consensus changes (timeouts, thresholds, slashing rules).

Governance ensures TAC can evolve securely while staying decentralized.

5. Use Cases & Applications(template)

TAC enables a range of hybrid applications with cross-chain interoperability that combine the user base and accessibility of The Open Network (TON) with the programmability and liquidity of the Ethereum ecosystem. This section outlines potential use cases and integration patterns across DeFi, consumer applications, and developer tools.

5.1 TON-Originated DeFi Interactions

Example:

- Curve pool interaction
- Staking TON into yield vaults
- Accessing tokenized assets from TON side

5.2 Hybrid Telegram MiniApps

Example:

- MiniApp game that stakes user rewards into a TAC DeFi vault
- Content app triggering a token unlock on EVM based on engagement
- Fiat ↔ token flow anchored in TAC contracts

5.3 Cross-Chain dApps and Proxies

Example:

- Single-instance Uniswap fork with dual-side access
- NFT contract accessible via TON wallet
- EVM contract with embedded callback to TON Adapter

5.4 Automation and Delegation

Example:

- Cross-chain voting systems
- Automated rebalancing strategies
- Subscription-based DeFi logic triggered from Telegram

5.5 Future Integrations

Example:

- TON-native identity primitives tied to EVM verification
- Consumer fintech flows embedded into TAC-based settlements

6. Roadmap

Timeline	Detailed Plan
2025 - Q2	April • TAC Summoning pre-mainnet liquidity program • Mainnet preparations for launch (inc. testing, stress testing, etc.) • Audits fix review finalization • Performance optimizations • Monitoring tools • Migration from mongo to postgre (sequencers layer) May • Series A fundraise • SOCs • Mainnet launch and monitoring • Veda TON Vaults integration • Agglayer, Layer0, USDT bridge integration via TON • MAINNET PRE-LAUNCH
	June • MAINNET LAUNCH • Merkl rewards system integration • Universal proxy for dApps with smart accounts support • Additional audits • TON network improvements integration • Initial version of sequencers network v1.5 with p2p support for mutisig opts
2025 - Q3	July • Advanced tooling for developers August - September • Test of Sequencer network intermediate upgrade (Sequencers network v.2) / Continue work on POS based Permissionless version of the sequencer network
2025 - Q4	• POS based Permissionless version of the sequencer network release • Audits and preparation for open-source process • Applying new TON features such as TON payment network and new APIs
2026	AltVM integration R&D

Conclusion

This whitepaper outlines the architecture and protocol mechanics of TAC, a Layer-1 compatibility chain purpose-built to connect The Open Network (TON) with the Ethereum ecosystem. Rather than relying on asset wrapping, third-party bridges, or middleware, TAC enables native interaction between TON wallets and EVM applications through deterministic, validator-secured message execution.

At the core of TAC's design is a system of stateless proxy contracts, decentralized sequencers, and on-chain Merkle root consensus that allows secure cross-chain communication without compromising user experience. Transactions are executed via standard EVM contracts, while users interact entirely from within the TON environment using their native tokens.

The protocol is structured for upgradeability, with economic security enforced via delegated proof-of-stake. All execution is validated through cryptographic proofs and governed via on-chain mechanisms, ensuring long-term adaptability and resilience.

TAC is a unified execution layer that integrates the user scale of Telegram with the composability of Ethereum. By abstracting away cross-chain complexity and aligning token incentives across chains, TAC enables a new class of hybrid applications—where users, developers, and dApps converge across different environments without friction.