The Theta Ledger

Theta Labs Dev Team

Contact: Jieyi Long (jieyi@thetatoken.org)

Introduction

The Theta Ledger is a decentralized ledger designed for the video streaming industry. It powers the Theta token economy which incentives end users to share their redundant bandwidth and storage resources, and encourage them to engage more actively with video platforms and content creators. To realize these goals, a number of challenges, many of which are unique for video streaming applications, need to be tackled.

One of such challenges is to support **ultra high transaction throughput**. Although many blockchain projects are facing transaction throughput problems, scaling for live video streaming is different and possibly even more complex. Typically, video segments are a couple seconds long. To achieve the finest granularity of a token reward — one micropayment per video segment — even a live stream with a moderate ten thousand concurrent viewers could generate a couple thousand microtransactions per second, which far exceeds the maximum throughput of today's public chains, such as Bitcoin and Ethereum. Popular live streams like major esport tournaments can attract more than one million viewers watching one stream simultaneously, not to mention multiple concurrent live streams, which could potentially push the required transaction throughput to the range of millions per second.

A byproduct of the high throughput is **rapidly growing storage consumption**. Storing the micropayments is highly storage demanding. With tens of thousands of transactions added to the ledger every second, the storage space of a normal computer could run out rather quickly.

Video streaming applications typically require **fast consensus**. For bandwidth sharing reward, the users that offer redundant bandwidth typically want the payment to be confirmed before sending the next one. Other use cases, such as virtual gift donations live stream hosts, also require short confirmation time to enable to real-time interaction between the hosts and the audience.

Last but not the least, as in any blockchain, security of the ledger is important. Security is highly correlated with the **level of decentralization**. In a Proof-of-Stake (PoS) based consensus mechanism, decentralization means even stake distribution among consensus participants. Ideally, the consensus mechanism should allow thousands of *independent* nodes, each with similar amount of stake, to participate in the block finalization process, and each has a local copy of the blockchain. To compromise such a system, a significant amount of independent nodes needs to be controlled by the attackers, which is difficult to achieve.

To achieve these goals, we have designed our PoS consensus algorithm based on the Byzantine Fault Tolerance (BFT) protocols, which offers good guarantees such as consistency (a.k.a. safety) when more than 2/3 of nodes running the ledger software are honest. However, the traditional BFT algorithms do not allow high level of decentralization. They typically incur $O(n^2)$ messaging complexity even for the normal (non-faulty proposer) case, where n is the number of nodes participated in the consensus protocol. When we have thousands of nodes, it will take considerable amount of time to reach agreement. In this paper, we will present a novel **multi-level BFT consensus mechanism** that allows mass participation, and yet able achieves 1000+ TPS throughput with the transaction confirmation time as short as a few seconds.

Such level of transaction throughput, although already much higher than Bitcoin and Ethereum, is still not sufficient to handle the micropayments for the "pay-per-byte" granularity. To further increase the throughput, the Theta Ledger provides native support for off-chain scaling, with a "resource oriented micropayment pool" which further amplifies the supportable throughput by several order of magnitudes.

We note that the off-chain payment support not only boosts the throughput, but also decreases the number of the transactions that need to be stored in the blockchain. On top of that, we introduce the technique of state and block history pruning to further reduce the storage space requirement. Moreover, we have adopted the microservice architecture for the storage system, which can adapt to different types of machines and storage backends, be it powerful server clusters running in data centers, or commodity desktop PCs.

We are actively working on a reference implementation of the Theta Ledger using Golang. The source code is available on GitHub: https://github.com/thetatoken/theta-protocol-ledger-2

The Consensus Mechanism

Multi-Level BFT

In this paper we propose a novel **multi-level BFT consensus mechanism** which allows thousands of nodes to participate in the consensus process, and yet can support very high transaction throughput (1000+ TPS).

The core idea is to have a small set of nodes, which forms the **validator committee**, to produce a chain of blocks as fast as possible using a PBFT-like[1] process. With sufficient number of validators (e.g. 10 to 20), the validator committee can produce blocks at a fast speed, and yet it is already difficult enough for the adversary to compromise. Hence, it is reasonable to expect that they will produce a chain of blocks without forks with very high probability. Then, all the thousands of consensus participants, called the **guardians**, can finalize the chain generated by the validator committee. Here “finalization” means to convince each honest guardian that more than 2/3 of all the other guardians see the same chain of blocks.

Since there are many more guardians than validators, it could take longer time for the guardians to reach consensus than the validator committee. In order for the guardians to finalize the chain of blocks at the same speed as the validator committee produces new blocks, the guardian nodes can process the blocks at a much coarser grain. To be more specific, they only need to agree on the hash of the **checkpoint blocks,** i.e. blocks whose height are a multiple of some integer T (e.g. T = 100). This **“leapfrogging” finalization strategy** leverages the immutability characteristic of the blockchain data structure -- as long as two guardian nodes agree on the hash of a block, with overwhelming probability, they will have exactly the same copy of the entire blockchain up to that block. Finalizing only the checkpoint blocks gives sufficient time for the thousands of guardians to reach consensus. Hence, with this strategy, the two independent processes, i.e., block production and finalization, can advance with the same pace.

Under the normal condition, finalizing a checkpoint block is similar to the "commit" step of the celebrated PBFT algorithm since each guardian has already stored the checkpoint block locally. Moreover, the checkpoint block has been signed by the validator committee, and hence highly likely that all the honest guardians have the same checkpoint. Thus, we only need a protocol for the honest guardians to confirm that indeed more than 2/3 of all guardians have the same checkpoint hash.

A naive all-to-all broadcasting of the checkpoint block hash could work, but it yields quadratic communication overhead, and so cannot scale to large number of guardians. Instead we propose an **aggregated signature gossip** scheme which could significantly reduce the messaging complexity. The core idea is rather simple. Each guardian node keeps combining the partially aggregated signatures from all its neighbors, and then gossip out this aggregated signature, along with a compact bitmap which encodes the list of signers. This way the signature share of each node

[1] *Castro et al*. Practical Byzantine Fault Tolerance

can reach other nodes at exponential speed thanks to the gossip protocol. Within $O(\log n)$ iterations, with high probability, all the honest guardian nodes should have a string which aggregates the signatures from all other honest nodes if there is no network partition. On the other hand, the signature aggregation keeps the size of the node-to-node messages small, and thus further reduces the communication overhead.

As mentioned above, the **validator committee** comprises of a limited set of validator nodes, typically in the range of ten to twenty. They can be selected through an election process, or a randomized process, and may subject to rotation to improve security. To be eligible to join the validator committee, a node needs to lock up a certain amount of stake for a period of time, which can be slashed if malicious behavior is detected. The blocks that the committee reaches consensus on are called **settled blocks**, and the process to settle the blocks is called the **block settlement process**.

The **guardian pool** is a *super set* of the validator committee, i.e. *a validator is also an guardian*. The pool contains a large number of nodes, which could be in the range of thousands. With a certain amount of token lockup for a period of time, any node in the network can instantly become an guardian. The guardians download and examine the chain of blocks generated by the validator committee and try to reach consensus on the the checkpoints with the above described “leapfrogging” approach. By allowing mass participation, we can greatly enhance the transaction security. The blocks that the guardian pool has reached consensus on are called **finalized blocks**, and the process to finalize the blocks is called the **block finalization process**.

The name **multi-level BFT consensus mechanism** reflects the fact that the validator/guardian division provides multiple levels of security guarantee. The validator committee provides the first level of protection — with 10 to 20 validators, the committee can come to consensus quickly. Yet it is resistant enough to attacks — in fact, it already provides similar level of security compared to the DPoS mechanism if each validator nodes is run by an independent entity. Thus, a transaction can already be considered safe when it has been included in a settled block, especially for low stake transactions. The guardian pool forms the second line of defense. With thousands of nodes, it is substantially more difficult for attackers to compromise, and thus provides a much higher level of security. In the unlikely event that the validator committee is fully controlled by attackers, the guardians can re-elect the validators, and the blockchain can restart advancing from the most recent block finalized by the guardians. A transaction is considered irreversible when it is included in a finalized block. We believe this mechanism achieves a good balance among transaction throughput, consistency, and level of decentralization, the three corners of the so-called “**impossible triangle**”.

The multi-level security scheme suits video streaming applications well. For streaming platforms, most of the transactions are micropayments (e.g. payment for peer bandwidth, virtual gifts to hosts, etc.) which typically have low value, but require fast confirmation. For such low stake payments, the users only need to wait for block settlement, which is very fast, in the matter of seconds. For high stake transfers, the user can wait longer until the block containing the transaction is finalized, which could take slightly longer time, but still in the range of minutes.

System Model

Before diving into the details of the block settlement and finalization process, we first list our assumptions of the system. For ease of discussion, **without loss of generality, below we assume each node (be it a validator or an guardian) has the same amount of stake**. Extending the algorithms to the general case where different nodes have different amount of stake is straightforward.

Validator committee failure model: There are m validator nodes in total. Most of the time, at most one-third of them are byzantine nodes. They might be fully controlled by attackers, but this happens only rarely. We also assume that between any pair of validator nodes there is a direct message channel (e.g. a direct TCP connection).

Guardian pool failure model: There are n guardian nodes in total. At any moment, at most one-third of them are byzantine nodes. We do not assume a direct message channel between any two guardians. Messages between them might need to be routed through other nodes, some of which could be byzantine nodes.

Timing model: We assume the "weak synchrony" model. To be more specific, the network can be asynchronous, or even partitioned for a bounded period of time. Between the asynchronous periods there are sufficient long period of time where all message transmissions between two honest nodes arrive within a known time bound Δ. As we will discuss later in the paper, during the asynchronous period, the ledger simply stops producing new blocks. It would never produce conflicting blocks even with network partition. During synchronous phases, block production will naturally resumes, and eventual liveness can be achieved.

Attacker model: We assume powerful attackers. They can corrupt a large number of targeted nodes, but no more than one-third of all the guardians simultaneously. They can manipulate the network at a large scale, and can even partition the network for a bounded period of time. Yet they are computationally bounded. They cannot forge fake signatures, and cannot invert cryptographic hashes.

The Block Settlement Process

Block settlement is the process that the validator committee reaches agreement and produce a chain of blocks for the guardian pool to finalize. Inspired by recent Proof-of-Stake research works including Tendermint[2], Casper FFG[3], and Hot-Stuff[4], we have designed and implemented the block settlement algorithm described below. It employs a rotating block proposer strategy where the validators take turns to propose new blocks. Then, the committee votes on the blocks to determine their order using a protocol similar to Casper FFG and Hot-Stuff.

Block Proposal

The validators rotate in a round robin fashion to play the role of block proposer, which is responsible for proposing the next block for the validator committee to vote on. To enable the round robin rotation, each proposer maintains a local logical clock called **epoch**. Assuming there are m validators, during epoch t, the validator with index $(t \mod m)$ is elected as the proposer for that epoch. We note it is important that

1) the epoch t should not stuck so the liveness of the proposer rotation is guaranteed; and
2) the epoch t of different validators should be mostly in sync, i.e. most of the time all the validators have the same *t* value, so they can agree on which node should produce the next block.

Below is our protocol for proposer election and block proposal.

[2] *Buchman et al*. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains
[3] *Buterin et al*. Casper the Friendly Finality Gadget
[4] *Yin et al*. HotStuff: BFT Consensus in the Lens of Blockchain

Protocol: Round Robin Block Proposal

```
$t \leftarrow 0$,  $proposer \leftarrow 0$
$voted \leftarrow$ false, $received \leftarrow$ false, $timeout \leftarrow$ false

loop begin
  $proposer \leftarrow t \bmod m$
  if ($proposer == self.index$) and (not proposed yet) begin // node elected as the proposer
    propose one block
  end

  $voted \leftarrow$ the node has proposed or voted for a block for epoch $t$
  $received \leftarrow$ the node has received $m/3 + 1$ $EpochChange(t + 1)$ messages
  $timeout \leftarrow$ timeout reached
  if $voted$ or $received$ or $timeout$ begin
    broadcast message $EpochChange(t + 1)$
  end

  if the node has received $2m/3$ $EpochChange(t + 1)$ messages begin
    $t \leftarrow t + 1$ // enters epoch $t + 1$
    $voted \leftarrow$ false, $received \leftarrow$ false, $timeout \leftarrow$ false
  end

  sleep for some time
end
```

***Figure 1**. The round robin block proposal protocol*

The protocol defines a message $EpochChange(t + 1)$, which can be viewed as a synchronization message passed among the validators to assist them to advance to the next epoch $t + 1$ together. Essentially, a validator **broadcasts** message $EpochChange(t + 1)$ to all other validators if any of the following conditions is met:

1) the node has proposed or voted for a block in epoch t, or
2) the node has received $m/3 + 1$ $EpochChange(t + 1)$ messages from other validators, or
3) the node timed out for epoch t (the timeout is set to 4Δ).

On the other hand, the validator **enters** epoch $t + 1$ when it has received $2m/3$ $EpochChange(t + 1)$ messages from other nodes.

Here we show that this protocol meets the above two requirements.

Eventual Progression: All the honest nodes will eventually enter epoch $t + 1$. In the worst case, all the honest nodes (at least $2m/3 + 1$ nodes) reach timeout and broadcast the $EpochChange(t + 1)$ messages. Under the timing model assumption, all these messages will be delivered within time Δ after being sent out. Thus each honest node will receive at least $2m/3$ $EpochChange(t + 1)$ messages, and it then enters epoch $t + 1$.

Epoch Synchrony: Intuitively, this means the epochs of all the honest nodes "move together". More precisely, we claim that the time any two honest nodes enter epoch $t + 1$ differ by at most most 2Δ. To prove this, we note that since there are at most f faulty nodes, for the first honest node to enter epoch $t + 1$, at least $m/3$ other honest nodes must have broadcasted the $EpochChange(t+1)$ messages. This honest node then also broadcasts an $EpochChange(t+1)$ message following the protocol. After at most Δ, any honest node should have received at least $m/3+1$ $EpochChange(t+1)$ messages, which triggers them to also broadcast the $EpochChange(t+1)$ message. After Δ, all the honest nodes receive $2m/3$ $EpochChange(t+1)$ messages and enter epoch $t+1$. Thus, at most 2Δ after the first honest node enters epoch $t+1$, the last honest node will enters the same epoch.

In practice, when the network latency is small enough, all the honest nodes should enter epoch $t+1$ at almost the same time. As a result, they can agree on who is the next proposer. Also we note that for the actual implementation, the $EpochChange(t+1)$ messages can be combined with other types of messages (e.g. block votes) to improve the efficiency. So that in the normal case (no proposer failure), no additional synchronization overhead is added to the system for epoch changes.

Block Consensus Among Validators

The protocol to settle proposed blocks involves a PBFT-based voting procedure among all validators, similar to Casper FFG and Hot-Stuff. In the Theta Ledger blockchain, the header of each block contains a hash pointer to its parent block (i.e. the previous block in the chain), similar to Bitcoin and Ethereum. Two blocks are conflicting if neither block is an ancestor of the other. If there are multiple, conflicting block proposals for the same epoch, an honest validator would keep all of them until one becomes settled, and then it discards all conflicting blocks.

The block settlement protocol operates epoch by epoch. The proposer for the current epoch sends to all validators a block proposal. A validator reacts by broadcasting a vote for the block. All messages are signed by their senders.

The header of the proposed block might carry a **commit-certificate**, which consists of at least $(2m/3+1)$ signed votes for its parent block. We note that under the assumption that no more than $m/3$ validators are faulty, at most one block per height can obtain a commit-certificate. A commit-certificate for a block indicates this block and all its predecessors are committed. The proposed block may carry no commit-certificate, if its parent block did not get $\geq 2m/3+1$ signed votes.

For the validators that are not the current proposer, their job is to vote on the proposed blocks. Once a validator receives the new block, it broadcasts a signed vote to all validators, so it can be collected by the proposer of the next epoch to form the commit-certificate. If **two consecutive blocks** A and B both receive a commit-certificate, then the parent block A and all its predecessors are considered **settled**. To ensure safety, we require the **honest nodes never vote for a block that conflicts with a settled block**. When there are forks (either due to faulty proposer or asynchrony), **the honest nodes should vote for the blocks on the longest fork**.

The figure below illustrates the block settlement process. Assume that the proposer for height 101 is faulty, and it proposed two conflicting blocks X_{101} and Y_{101}, which leads to two branches. Assuming neither block X_{101} nor Y_{101} gets $\geq 2m/3+1$ votes, then, neither the header of X_{102} nor Y_{102} contains the commit-certificate (denoted by nil in the figure). However, at some point branch X grows faster, and two consecutive blocks X_{102} and X_{103} both obtain $\geq 2m/3+1$ votes. After that the upper branch X up to block X_{102} is considered settled. And the lower branch Y can be discarded.

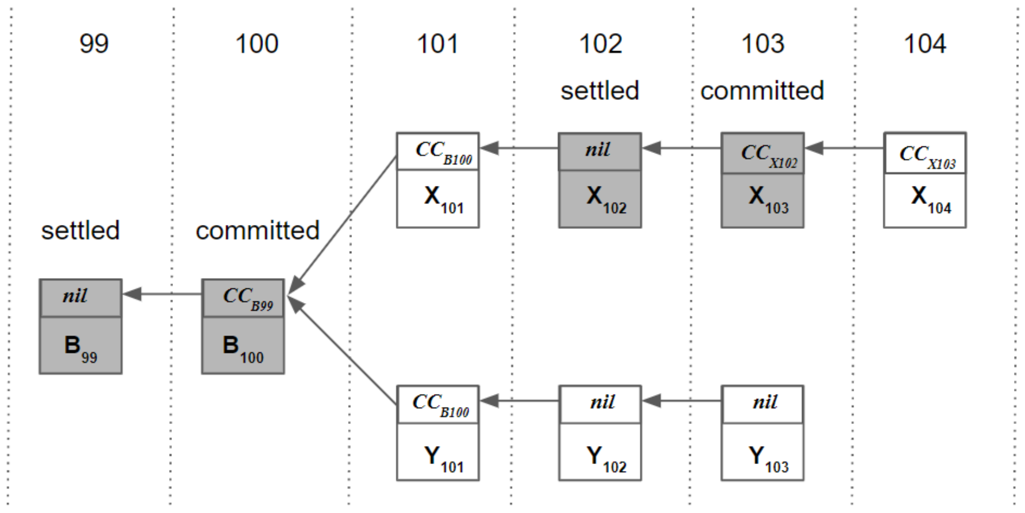


Figure 2. *The block settlement process*

The above example also illustrates one advantage of our implementation compared to other PBFT based protocol like Tendermint — a block that does not receive commit-certificate can also be included in the settled chain, as long as one of its successor blocks is settled. For instance, block X_{101} in the example did not get a commit-certificate, but after block X_{102} is settled, it is also considered settled. This reduces the waste of computation power and helps increase the transaction throughput.

Analysis

Safety: Safety means all honest validators agree on the same chain of blocks. More precisely, if one honest validator accepts a block A, then any future blocks accepted by other honest validators will appear in a chain of blocks that already contains A. The argument for safety is similar to Casper FFG and Hot-Stuff and is omitted here. We just want to point out that safety stems from the requirement that honest nodes never vote for a block that conflicts with a settled block.

Liveness: Liveness means the validator committee always makes progress, i.e., always able to produce and agree on new blocks. Here we show that under our timing model, during the synchronous periods, the committee can always achieve the liveness goal. First, in the “Block Proposal” section, we have proved that the epoch can always advance, and all the honest validators march forward together. In an epoch where the proposer is an honest validator, it will propose a new block. For the block settlement process, liveness depends on that during the synchronous periods, there are infinitely many epochs where two proposers in a row are honest, and wait sufficiently long to form the commit-certificate. We note this is guaranteed to happen infinitely often with the round robin rotation, since at least 2/3 of the validators are honest.

Transaction throughput: With ten to twenty validators, the committee can produce and settle the chain of block rather quickly. Average block production and settlement time is in the order of seconds, and this leads to high throughput as much as 1000+ transactions per second.

The Block Finalization Process

In this section, we will discuss the “leapfrogging” block finalization process in detail. As mentioned above, the guardians only need to reach consensus on the hashes of the checkpoint blocks, which are the blocks whose heights are multiple of of some integer T (e.g. T = 100).

To see why it is sufficient to finalize just the checkpoint blocks, we note that the transaction execution engine of the blockchain software can be viewed as a “deterministic state machine”, whereas a transaction can be viewed as a deterministic state transfer function. If two nodes run the same state machine, then from an identical initial state, after executing the same sequence of transactions, they will reach an identical end state. Note that this is true even when some of the transactions are invalid, as long as those transactions can be detected by the state machine and skipped. For example, assume there is a transaction tries to spend more tokens than the balance of the source account. The state machine can simply skip this transaction after performing the sanity check. This way the “bad” transactions have no impact on the state.

In the context of blockchain, if all the honest nodes have the same copy of the blockchain, they can be ensured to arrive at the same end state after processing all the blocks in order. But with one caveat — the blockchain might contain huge amount of data. How can two honest nodes compare whether they have the same chain of blocks efficiently?

Here the immutability characteristic of the blockchain data structure comes to the rescue. Since the header of each block contains the hash of the previous block, as long as two nodes have the same hash of the checkpoint block, with overwhelming probability, they should have an identical chain of blocks from genesis up to the checkpoint. Of course each guardian node needs to verify the integrity of the blockchain. In particular, the block hash embedded in each block header is actually the hash of the previous block. We note that a node can perform the integrity checks on its own, no communication with other nodes is required.

Interestingly, the immutability characteristic also enhances the tolerance to network asynchrony or even partition. With network partition, the guardians may not be able to reach consensus on the hash of a checkpoint. However, after the network is recovered, they can move on to vote on the next checkpoint. If they can then reach agreement, then all the blocks up to the next checkpoint are finalized, regardless of whether or not they have consensus on the current checkpoint.

To provide byzantine fault tolerance, an honest node needs to be assured that at least two-third of the guardians has the same checkpoint block hash. Hence it needs to receive signatures for a checkpoint hash from at least two-third of all guardians before the node can mark the checkpoint as finalized. This is to ensure safety, which is similar to the “commit” step in the celebrated PBFT protocol.

Since the guardians only need to vote on checkpoint hashes every T blocks, they have more time to reach consensus. A straightforward implementation of checkpoint finalization is thus to follow the PBFT “commit” step where each guardian broadcast its signature to all other guardians. This requires each node to send, receive and process $O(n)$ messages, where each message can be a couple kilobytes long. Even with T blocks time, this approach still cannot scale beyond a couple hundred guardian nodes, unless we select a large T value, which is undesirable since it increases the block finalization latency.

Scale to Thousands of guardians

To reduce the communication complexity and scale to thousands of guardians, we have designed an **aggregated signature gossip** scheme inspired by the BLS signature aggregation technique[5] and the gossip protocol. The scheme requires each guardian node to process a much smaller number of messages to reach consensus, which is much more practical. Below are the steps of the aggregated signature gossip protocol. It uses the BLS algorithm for signature aggregation.

Protocol: Aggregated Signature Gossip

```
$finalized \leftarrow$ false, $\sigma_i \leftarrow \mathbf{SignBLS}(sk_i, height_{cp} \parallel hash_{cp})$, $c_i \leftarrow \mathbf{InitSignerVector}(i)$
for $l = 1$ to $L$ begin
    send $(\sigma_i, c_i)$ to all its neighboring guardians
    if $finalized$ break
    wait for $(\sigma_j, c_j)$ from all neighbors until timeout
    verify each $(\sigma_j, c_j)$, discard if it is invalid
    aggregate valid signatures $\sigma_i \leftarrow \sigma_i \cdot \prod_j \sigma_j$, $c_i \leftarrow \left(c_i + \sum_j c_j\right) \bmod p$
    calculate the number of unique signers $s \leftarrow \sum^n I\,(c_i[k] > 0)$
    if $s \geq \frac{2}{3}n$ $finalized \leftarrow$ true
end
```

Figure 3. *The aggregated signature gossip protocol*

The core idea is rather simple. Each guardian node keeps combining the partially aggregated signatures from its neighbors, and then gossip this newly aggregated signature out. This way the signature share of each node can reach other nodes at exponential speed thanks to the gossip protocol. On the other hand, the signature aggregation keeps the size of the messages small, and thus reduces the communication overhead.

In the above diagram, i is the index of the current guardian node. The first line of the protocol uses function **SignBLS**() to generate its initial aggregated signature σ_i. It essentially signs a message which is the concatenation of the height and hash of the checkpoint block using the BLS signature algorithm, with multiplicative cyclic group G of prime order p, and generator g:

$$h_i \leftarrow H\left(pk_i,\ height_{cp} \parallel hash_{cp}\right)$$

$$\sigma_i \leftarrow (h_i)^{sk_i}$$

In the first formula above, function $H : G \times \{0, 1\}^* \rightarrow G$ is a hash function that takes both the public key pk_i and the message as input. This is to prevent the rogue public-key attack[6].

[5] *Boneh et al*. A Survey of Two Signature Aggregation Techniques
[6] *Boneh et al*. BLS Multi-Signatures With Public-Key Aggregation

The protocol also uses function **InitSignerVector()** to initialize the **signer vector** c_i, which is a n dimensional integer vector whose jth entry represents how many times the jth guardian has signed the aggregated signature. After initialization, its ith entry is set to 1, and the remaining entries are all set to 0.

After initialization, the guardian enters a loop. In each iteration, the guardian first sends out its current aggregated signature σ_i and the signer vector c_i to all its neighbors. Then, if it has not considered the checkpoint as finalized, it waits for the signature and signer vector from all its neighbors, or wait until timeout. Upon receiving all the signature and signer vectors, it checks the validity of (σ_j, c_j) using the BLS aggregated signature verification algorithm.

$$h_u \leftarrow H(pk_u,\ height_{cp} \,\|\, hash_{cp})$$

$$\text{check if } e\left(\sigma_j,\ g\right) \ = \ \prod_{u}^{n} (e\,(h_u,\ pk_u))^{c_j[u]}$$

where $e : G \times G \rightarrow G_T$ is bilinear mapping function from $G \times G$ to G_T, another multiplicative cyclic group also of prime order p. All the invalid signatures and their associated signer vectors are **discarded** for the next aggregation step. It is worth pointing out that besides $height_{cp}$, $hash_{cp}$, the above check also requires the public key pk_u of the relevant guardians as input. All these information should be available locally, since when an guardian locked up its stakes, its public key should have been attached to the stake locking transaction which has already been written into the blockchain. Hence, no communication with other nodes is necessary to retrieve these inputs.

The aggregation step aggregates the BLS signature σ_j, and updates the signer vector c_j. Note that for the vector update, we take $mod\, p$ for each entry. We can do this because $e\,(h_u,\ pk_u) \in G_T$, which is a multiplicative cyclic group of prime order p. This guarantees that the entries of vector c_j can always be represented with a limited number of bits.

$$\sigma_i \leftarrow \sigma_i \cdot \prod_{j} \sigma_j\,,\ c_i \leftarrow \left(c_i + \sum_{j} c_j\right) mod\, p$$

The algorithm then calculates the number of unique signers of the aggregated signature.

$$s \leftarrow \sum^{n} I\,(c_i\,[k] > 0)$$

Here function I: $\{\text{true, false}\} \rightarrow \{1,\ 0\}$ maps a true condition to 1, and false to 0. Hence the summation counts how many unique signers have contributed to the aggregated signature. If the signature is signed by more than two-third of all the guardians, the guardian considers the checkpoint to be finalized.

If the checkpoint is finalized, the aggregated signature will be gossipped out in the next iteration. Hence within $O(log(n))$ iterations all the honest guardians will have an aggregated signature that is signed by more than two-third of all the guardians if the network is not partitioned.

The loop has L iterations, L should be in the order of $O(log(n))$ to allow the signature to propagate through the network.

Analysis

Safety: Safety of the block finalization is easy to prove. Under the 2/3 supermajority honesty assumption, If two checkpoint hashes for the same height both get aggregated signatures from at least 2/3 of all guardians, at least one honest guardian has to sign different hashes for the same height, which is not possible.

Liveness: Without network partition, as long as L is large enough, it is highly likely that after $O(log(n))$ iteration, all the honest nodes will see an aggregated signature that combines the signatures of all honest signers. This is similar to how the gossip protocol can robustly spread a message throughout the network in $O(log(n))$ time, even with up to ⅓ byzantine nodes. When there is network partition, consensus for a checkpoint may not be able to reach. However, after the network partition is over, the guardian pool can proceed to finalize the next checkpoint block. If consensus can then be reached, all the blocks up to the *next* checkpoint are considered finalized. Hence the finalization process will progress eventually.

Messaging Complexity: The aggregated signature gossip protocol runs for L iterations, which is in the order of $O(log(n))$. In each iteration, the guardian needs to send message (σ_i, c_i) to all its neighboring guardians. Depending on the network topology, typically it is reasonable to assume that for an average node, the number of neighboring nodes is a constant (i.e. the number of neighbors does not grow as the total number of nodes grows). Hence the number of message a node needs to send/receive to finalize a checkpoint is in the order of $O(log(n))$, which is much better than the $O(n)$ complexity in the naive all-to-all signature broadcasting implementation. We do acknowledge that each message between two neighboring guardians contains an n dimensional signer vector c_i, where each entry of c_i is an integer smaller than prime p. However, we note that this vector can be represented rather compactly since most of its entries are small integers ($\ll p$) in practice.

To get a more concrete idea of the messaging complexity, let us work out an example. Assume that we pick a 170-bit long prime number p for the BLS signature, which can provide security comparable to that of a 1024-bit RSA signature. And there are 1000 guardians in total. Under this setting, c_i can be represented with about twenty kilobytes without any compression. Since most of the entries of c_i are far smaller than p, c_i can be compressed very effectively to a couple kilobytes long. Plus the aggregated signature, the size of each message is typically in the kilobytes range. Moreover, if we assume on average an guardian connects to 20 other guardians, then L can be as small as 5 (more than twice of $log_{20}(1000) = 2.3$). This means finalizing one checkpoint just requires an guardian to send/receive around 100 messages to/from its neighbors, each about a couple kilobytes long. This renders the aggregated signature gossip protocol rather practical to implement and can easily scale to thousands of guardian nodes.

Reward and Penalty for Validators and Guardians

The token reward and penalty structure is essential to encourage nodes to participate in the consensus process, and not to deviate from the protocol.

Both the validators and guardians can obtain a token reward. Each block includes a special Coinbase transaction that deposits newly minted tokens to the validator and guardian addresses. All the validators can get a share of tokens for each block. For guardians, rewarding every guardian for each block might not be practical since their number is large. Instead, we propose the following algorithm to randomly pick a limited number of guardians as the reward recipient for each block.

Denote the height of the newly proposed block by l, and cp is the most recently finalized checkpoint. The proposer should have received the aggregated signature σ_{cp} and corresponding signer vector c_{cp} for checkpoint cp. Upon validating (σ_{cp}, c_{cp}), the proposer can check the following condition for each guardian whose corresponding entry in vector c_{cp} is not zero (i.e. that guardian signed the checkpoint)

$$H(pk_i, \sigma_{cp} \| B_{l-1}) \leq \tau$$

where B_{l-1} is the hash of the block with height $l-1$, and $H : G \times \{0, 1\}^* \rightarrow G$ is the same hash function used in the BLS signature algorithm. If the inequality holds, the proposer adds the guardian with public key pk_i to the Coinbase transaction recipient list. Threshold τ is chosen properly such that only a small number of guardians are included. The proposer should also attach (σ_{cp}, c_{cp}) to the Coinbase transaction as the proof for the reward.

The Theta ledger also enforces token penalty should any malicious behavior be detected. In particular, if a block proposer signs conflicting blocks for the same height, or if a validator votes for different blocks for the same height, they should be penalized. Earlier we mentioned that to become either a validator or an guardian, a node needs to lock up a certain amount of tokens for a period of time. The penalty will be deducted from their locked tokens. The node that detects the malicious behavior can submit a special Slash transaction to the blockchain. The proof of the malicious behavior (e.g. signatures for conflicting blocks) should be attached to the Slash transaction. The penalty tokens will be pulled from the malicious node and awarded to the node that submitted the first Slash transaction.

In the unlikely event that more than one-third of the validators are compromised, the malicious validators can attempt to perform the double spending attack by forking the blockchain from a block that is settled but not yet finalized. However, this is detectable by the guardian pool, since forking will generate multiple blocks with the same height, but signed by more than two-third of the validators. In this case, the validators conducted double signing will be penalized, and the entire validator committee will be re-elected. After the validator committee is reinstated, the blockchain can continue to advance from the most recent finalized checkpoint.

Off-Chain Micropayment Support

As discussed in the introduction section, support for high transaction throughput is a must for a video streaming focused blockchain. We build the support for off-chain payment directly into the ledger to facilitate high volumes of transactions.

Resource Oriented Micropayment Pool

We have designed and implemented an off-chain “**Resource Oriented Micropayment Pool**” that is purpose-built for video streaming. It allows a user to create an off-chain micropayment pool that any other user can withdraw from using off-chain transactions, and is double-spend resistant. It is much more flexible compared to off-chain payment channels. In particular, for the video streaming use case, it allows a viewer to pay for video content pulled from multiple caching nodes without on-chain transactions. By replacing on-chain transactions with off-chain payments, the built-in “Resource Oriented Micropayment Pool” significantly improves the scalability of the blockchain.

The following scenario and diagram provide a comprehensive walkthrough of how the Resource Oriented Micropayment Pool works in application.

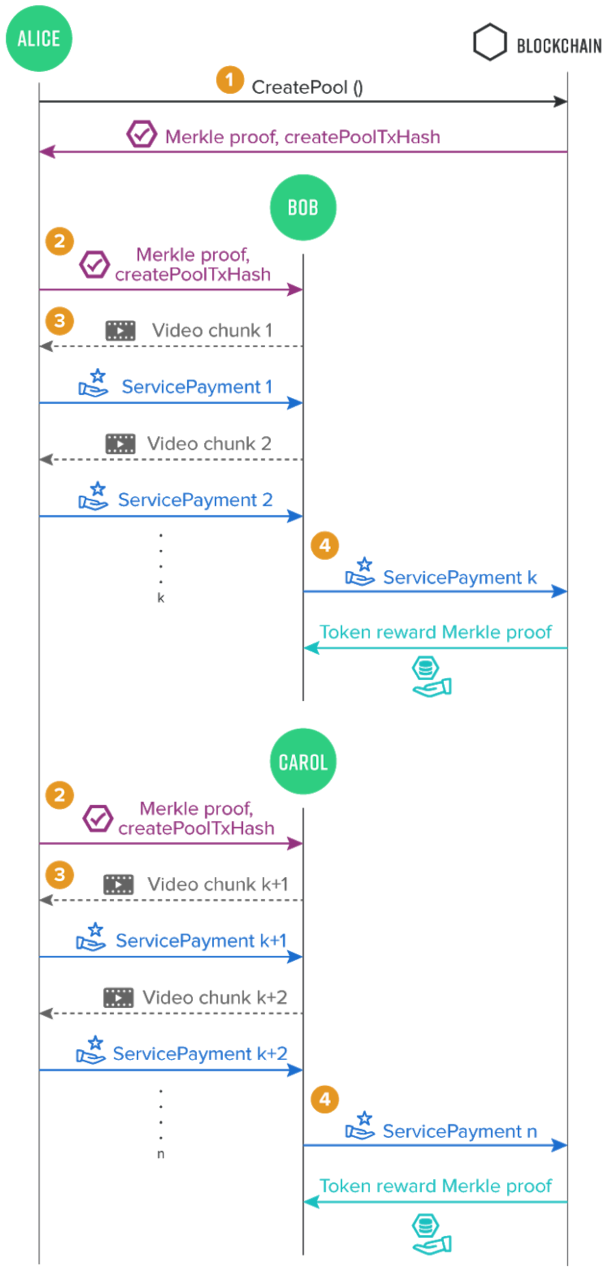


Figure 4. Resource Oriented Micropayment Pool *shows viewer Alice making off-chain transactions to cachers Bob and Carol for video chunks*

- **Step 1. Micropayment pool creation**: As the first step, Alice publishes an on-chain transaction to create a micropayment pool with a time-lock and a slashable collateral.

$$CreatePool(resourceId, deposit, collateral, duration)$$

A couple things to be noted. To create the pool, Alice needs to specify the “Resource ID” $resourceId$ that uniquely represents the digital content she intends to retrieve. It may refer to a video file, or a live stream.

The $deposit$ amount needs to be at least the total value of the resource to be retrieved. For instance, if the resource is a video file which is worth 10 tokens, then the deposit has to be at least 10 tokens.

The *collateral* is required to discourage Alice from double spending. If a double spending attempt from Alice is detected by the validators of the blockchain, the collateral will be slashed. Later in the blogpost we will show that if $collateral > deposit$, the net return of a double spend is always negative, and hence any rational user will have no incentive to double spend.

The duration is a time-lock similar to that of a standard payment channel. Any withdrawal from the payment pool has to be before the time-lock expires.

The blockchain returns Alice the Merkle proof of the *CreatePool*() transaction after it has been committed to the blockchain, as well as *createPoolTxHash*, the transaction hash of the *CreatePool*() transaction.

- **Step 2. Initial handshake between peers**: Whenever Alice wants to retrieve the specified resource from a peer (Bob, Carol, or David, etc.). She sends the Merkle proof of the on-chain *CreatePool*() transaction to that peer. The recipient peer verifies the Merkle proof to ensure that the pool has sufficient deposit and collateral for the requested resource, and both parties can proceed to the next steps.

- **Step 3. Off-chain micropayments**: Alice signs *ServicePayment* transactions and sends them to the peers off-chain in exchange for parts of the specified resource (e.g. a piece of the video file, a live stream segment, etc.). The *ServicePayment* transaction contains the following data:

$$targetAddress,\ transferAmount,\ createPoolTxHash,\ targetSettlementSequence,$$
$$\textbf{Sign}(SK_A,\ targetAddress \,\|\, transferAmount \,\|\, createPoolTxHash \,\|\, targetSettlementSequence)$$

  The *targetAddress* is the address of the peer that Alice retrieves the resource from, and the *transferAmount* is the amount of token payment Alice intends to send. The *targetSettlementSequence* is to prevent a replay attack. It is similar to the “nonce” parameter in an Ethereum transaction. If a target publishes a *ServicePayment* transaction to the blockchain (see the next step), its *targetSettlementSequence* needs to increment by one.

  The recipient peer needs to verify the off-chain transactions and the signatures. Upon validation, the peer can send Alice the resource specified by the *CreatePool*() transaction.

  Also, we note that the off-chain *ServicePayment* transactions are sent directly between two peers. Hence there is no scalability bottleneck for this step.

- **Step 4. On-chain settlement**: Any peer (i.e. Bob, Carol, or David, etc) that received the *ServicePayment* transactions from Alice can publish the signed transactions to the blockchain anytime before the timelock expires to withdraw the tokens. We call the *ServicePayment* transactions that are published the “on-chain settlement” transactions.

  Note that the recipient peers needs to pay for the gas fee for the on-chain settlement transaction. To pay less transaction fees, they would have the incentive to publish on-chain settlements only when necessary, which is beneficial to the scalability of the network.

We note that no on-chain transaction is needed when Alice switches from one peer to another to retrieve the resource. In the video streaming context, this means the viewer can switch to any caching node at any time without making an on-chain transaction that could potentially block or delay the video stream delivery. As shown in the figure, in the event that Bob leaves, Alice can switch to Carol after receiving k chunks from Bob, and keep receiving video segments without an on-chain transaction.

Moreover, the total amount of tokens needed to create the micropayment pool is $(collateral + deposit)$, which can be as low as twice of the value of the requested resource, no matter how many peers Alice retrieves the resource from. Using computational complexity language, the amount of reserved token reduces from $O(n)$ to $O(1)$ compared to the unidirectional payment channel approach, where *n* is the number of peers Alice retrieves the resource from.

Double Spending Detection and Penalty Analysis

To prevent Alice, the creator of the micropayment pool from double spending, we need to 1) be able to detect double spending, and 2) ensure that the net value Alice gains from double spending is strictly negative.

Detecting double spending is relatively straightforward. The validators of the Theta Network check every on-chain transaction. If a remaining deposit in the micropayment pool cannot cover the next consolidated payment transaction signed by both Alice and another peer, the validators will consider that Alice has conducted a double spend.

Next, we need to make Alice worse off if she double spends. This is where the collateral comes in. Earlier, we mentioned that the amount of collateral tokens has to be larger than the deposit. And here is why.

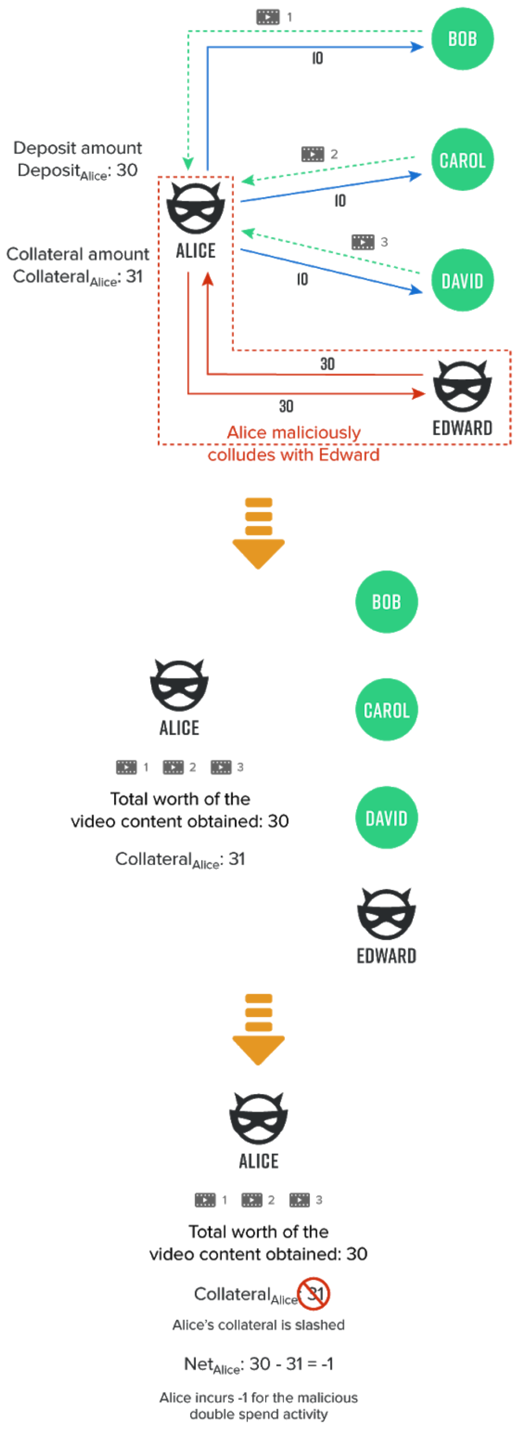
In Figure 5 below, Bob, Carol, and David are honest. Alice is malicious. Even worse, she colludes with another malicious peer Edward. Alice exchanges partially signed transactions with Bob, Carol, and David for the specified resource. Since Alice gains no extra value for the duplicated resource, the maximum value she gets from Bob, Carol, and David is at most the *deposit* amount. As Alice colludes with Edward, she can send Edward the full $deposit$ amount. She then asks Edward to commit the settlement transaction before anyone else and return her the $deposit$ later. In other words, Alice gets the resource which is worth at most the $deposit$ amount for free, before the double spending is detected. Later when Bob, Carol, or David commit the settlement transaction, the double spending is detected, and the full $collateral$ amount will be slashed. Hence, the net return for Alice is

$$net_{Alice} = deposit - collateral$$

Therefore, we can conclude that for this scenario, as long as $collateral > deposit$, Alice's net return is negative. Hence, if Alice is rational, she would not have any incentive to double spend.

We can conduct similar analysis for other cases. The details are omitted here, but it can be shown that in all cases Alice's net return is always negative if she conducts a double spend.

Another case is that Alice is honest, but some of her peers are malicious. After Alice sends a micropayment to one of those peers, it might not return Alice the resource she wants. In this case, Alice can turn to another peer to get the resource. Since each incremental micropayment can be infinitesimally small in theory, Alice's loss can be made arbitrarily small.



1
BOB
10
Deposit amount
Deposit$_{Alice}$: 30
2
CAROL
10
ALICE
3
Collateral amount
Collateral$_{Alice}$: 31
10
DAVID
30
30
EDWARD
Alice maliciously colludes with Edward
BOB
ALICE
CAROL
1 2 3
Total worth of the video content obtained: 30
DAVID
Collateral$_{Alice}$: 31
EDWARD
ALICE
1 2 3
Total worth of the video content obtained: 30
Collateral$_{Alice}$: 31
Alice's collateral is slashed
Net$_{Alice}$: 30 - 31 = -1
Alice incurs -1 for the malicious double spend activity

***Figure 5**. **Malicious Actor Detection and Penalty** shows malicious actor Alice attempting to make a double spend and the resulting penalty she receives*

Ledger Storage System

Using a public ledger to facilitate the micropayments for streaming is challenging, not only because high transaction throughput, but also for storage space management. To achieve the “pay-per-byte” granularity, each viewer could send out a payment every few seconds. With even a moderate ten thousand concurrent users, it could generate a couple thousands of transactions per second. Even with the off-chain payment pool which already reduces the amount of on-chain transactions dramatically, the block and state data could still balloon rather quickly.

We have designed a storage system that addresses this problem, and can adapt to different types of machines, be it a powerful server cluster running in data centers, or a commodity desktop PC.

Storage Microservice Architecture

To harness the processing and storage power of server clusters, the key design decision is to adopt the popular microservice architecture commonly seen for modern web service backends, where different modules of the ledger can be configured to run on different machines. In particular, the consensus module and the storage module can be separated. Potentially the consensus module can run on multiple machines using the MapReduce framework to process transactions in parallel.

The Theta Ledger stores both the transaction blocks and the account state history, similar to Ethereum. The bottom layer of the storage module is a key value store. The Theta Ledger implements the interfaces for multiple databases, ranging from single machine LevelDB to cloud based NoSQL database such as MongoDB, which can store virtually unlimited amount of data. Thus the ledger can run on one single computer, and can also be configured to run on server clusters.

History Pruning

While the microservice architecture suites the powerful server clusters well, we still face storage space constraints when running the ledger on a lower-end home PC. We have designed several techniques to reduce the storage consumption.

Similar to Ethereum, the Theta Ledger stores the entire state for each block, and the state tree root is saved in the header of the corresponding block. To reduce the space consumed by the state history, the Theta Ledger implements state history pruning, which leverages a technique called reference counting illustrated in the figure below.

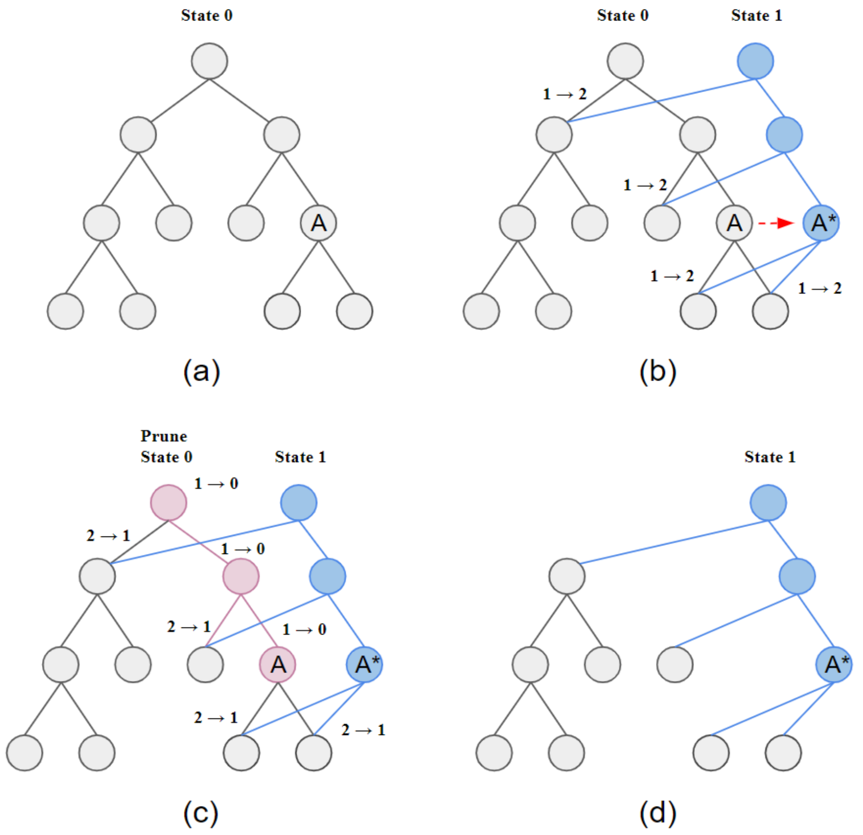


***Figure 6**. State history pruning with reference counting*

The ledger state (i.e. the token balance of each account, etc.) is stored using a Merkle-Patricia trie. Figure 6(a) depicts the initial state tree, whose root is denoted by $\mathrm{State}\ 0$. Each node is the tree has an attribute called the "reference count", which is equal to the number of parents of the node. In the initial state tree, each node has only one parent, so the reference count are all set to 1.

In Figure 6(b), account A is updated to A^* after applying the transactions in the newly settled block. Hence a new Merkle state root $\mathrm{State}\ 1$ is created, along with the Merkle branch connecting the new root $\mathrm{State}\ 1$ and A^* (the blue nodes and edges). Since new nodes are added, we update the reference count of direct children of these new nodes from 1 to 2.

At some point we decided to delete $\mathrm{State}\ 0$ to save some storage space. This is done by deleting the nodes whose reference count is zero recursively starting from the root $\mathrm{State}\ 0$, until no node can be deleted. Whenever a node is deleted, the reference count of all its children will be decremented by one. Figure 6(c) illustrates the process, and Figure 6(d) shows the result of the pruning. To achieve the maximum level to state storage compaction, once a block is finalized by the guardian pool, we can delete all the history prior to that block. The ledger can also be configured to keep a limited history of states, for example, the state trees of the latest 1000 blocks, depending on the available storage space.

It can be shown that with the reference counting technique, pruning a state tree has the time complexity of $O(k \log N)$, where k is the number of accounts updated by the transactions in one block, and N is the total number of accounts. Typically, k is in the range of a couple hundreds to a thousand. Hence, pruning a state tree should be pretty efficient and should not take much time.

Managing the space consumed by the transaction blocks is even simpler, after a block is finalized, we can simply delete all its previous blocks, or keep a limited history similar to the state trees.

With these techniques, common PCs and laptops are sufficient to run the guardian nodes.

State Synchronization

One of the pain points using earlier generation blockchains is the state synchronization time. After spinning up a new node, typically it needs to download the full block history all the way from the genesis block. This could take days to complete, and already becomes a hurdle for user adoption.

The state and block history stored by the full nodes can help reduce the synchronization time dramatically. After a new node start, the first step is to download all the validator and guardian join/leave transactions and the headers of the blocks that contain these special transaction up to the latest finalized block. With these special transactions and the headers which contain the validator and guardian signatures, the new node can derive the current validator committee and guardian pool. Since the validator and guardian set changes are relatively infrequent, the amount of data need to be downloaded and verified for this step should be minimal.

In the second step, the new node downloads the state tree corresponding to the latest finalized block. And it needs to confirm that the root hash of the tree equals the state hash stored in the latest finalized block. Finally, the new node verifies the integrity of the state tree (e.g. the validity of the Merkle branches). If all the checks are passed, the new node can start listening to new blocks and start participating in the consensus process.

Conclusions

In this paper, we have introduced the Theta ledger, a decentralized ledger system designed for the video streaming industry. It employs a novel multi-level BFT consensus engine, which supports high transaction throughput, fast block confirmation, and yet allows mass participation in the consensus process. Moreover, off-chain payment support is built directly into the ledger through the resource-oriented micropayment pool, which is designed specifically to achieve the “pay-per-byte” granularity for streaming use cases. Finally, the ledger storage system leverages the microservice architecture and reference counting based history pruning techniques, and is thus able to adapt to different computing environment, ranging from high-end data center server clusters to commodity PCs and laptops.