Loom: A new architecture for a high performance blockchain v0.8.5

Anatoly Yakovenko
anatoly@loomprotocol.com

Abstract

This paper proposes a Proof of History (PoH) - a proof for verifying passage of time between events. PoH is used to encode passage of time into a ledger - an append only data structure. When used alongside a consensus algorithm such as Proof of Work (PoW) or Proof of Stake (PoS), PoH can reduce messaging overhead in a Byzantine Fault Tolerant replicated state machine. This paper proposes two algorithms that leverage the time keeping properties of the PoH ledger - a Proof of Stake algorithm that can recover from partitions of any size and an efficient streaming Proof of Replication (PoRep). The combination of PoRep and PoH provides a defense against forgery of the ledger with respect to time and storage. The protocol is analyzed on a 1gbps network, and this paper shows that throughput up to 710k transactions per second is possible with today's hardware.

1 Introduction

Blockchain is an implementation of a fault tolerant replicated state machine. Current publicly available blockchains do not rely on time, or make a weak assumption about the participant's abilities to keep time [4]. Each node in the network usually relies on their own local clock without knowledge of any other participants clocks in the network. The lack of a trusted source of time means that when a message timestamp is used to accept or reject a message, there is no guarantee that every other participant in the network will make the exact same choice. The PoH presented here creates a ledger with verifiable passage of time, duration between events and message ordering.

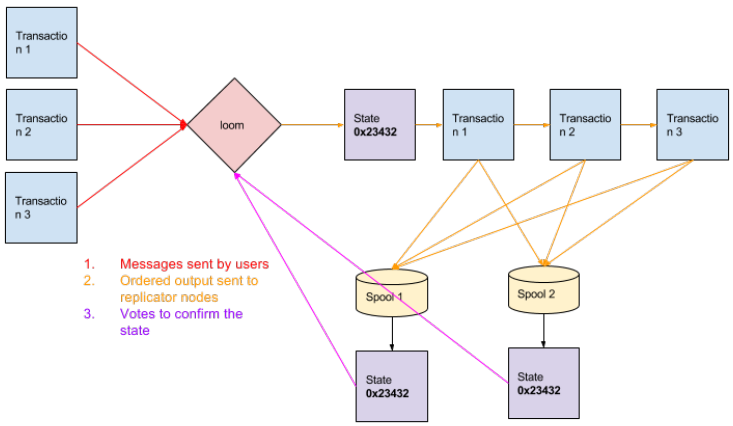


Figure 1: Transaction flow throughout the network.

Every node in the network can rely on the recorded passage of time in the ledger without trust.

2 Outline

The remainder of this article is organized as follows. Overall system design is described in Section 3. In depth description of Proof of History is described in Section 4. In depth description of the proposed Proof of Stake consensus algorithm is described in Section 5. In depth description of the proposed fast Proof of Replication is described in Section 6. System Architecture and performance limits are analyzed in Section 7. A high performance GPU friendly smart contracts engine is described in Section 7.5

3 Design

As shown in Figure 1, Loom generates a Proof of History. It sequences user messages. It orders them such that they can be efficiently processed by maximizing throughput. It executes the transactions on the current state that is stored in RAM. It publishes the transactions and a signature of the state to the replications nodes called Spools. Spools execute the same transactions on their copies of the state, and publish their computed signatures of the state as confirmations. The published confirmations serve as votes for the consensus algorithm.

There are many Looms in the network, and each Spool node has the same hardware capabilities as a Loom and can be elected as a Loom. A Primary and lower ranked Looms are elected via a consensus algorithm. Elections for the proposed PoS algorithm are covered in depth in Section 5.6.

The Loom generates a Proof of History sequence, which along with global read consistency also provides a verifiable passage of time for the network.

In terms of CAP theorem, Consistency is almost always picked over Availability in an event of a Partition. In case of a large partition, this paper proposes a mechanism to eventually recover control of the network from a partition of any size. This is covered in depth in Section 5.10.3.

4 Proof of History

Proof of History provides a way to cryptographically verify passage of time between two events. It uses a cryptographically secure function whose output cannot be predicted from the input, and must be completely executed to generate the output. The function is run in a sequence, its previous output as the current input, periodically recording the current output, and how many times it's been called. The output can then be re-computed and verified by external computers in parallel by checking each period in parallel on a separate core. Data can be timestamped into this sequence by appending the data into the state of the function. The recording of the state, index and data as it was appended into the sequences provides a timestamp that guarantees that the data was created sometime before the next hash was generated in the sequence. Horizontal scaling is also possible as multiple generators can synchronize amongst each other by mixing their state into each others sequences. Horizontal scaling is discussed in depth in Section 4.4

4.1 Description

With a cryptographic function, like a cryptographic hash (`sha256`, `md5`, `sha-1`), whose output cannot be predicted without running the function, run the function from some random starting value and take its output and pass it as the input into the same function again. Record the number of times the function has been called and the output at each call. The starting random value chosen could be any string, like the headline of the New York times for the day.

For example:

PoH Sequence		
Index	Hash	Operation
1	hash1	sha256("any random starting value")
2	hash2	sha256(hash1)
3	hash3	sha256(hash2)

Where `hashN` represents the actual hash output.

It is only necessary to publish a subset of the hashes and indices at an interval.

For example:

PoH Sequence		
Index	Hash	Operation
1	hash1	sha256("any random starting value")
200	hash200	sha256(hash199)
300	hash300	sha256(hash299)

As long as the hash function chosen is collision resistant, this set of hashes can only be computed in sequence by a single computer thread, because there

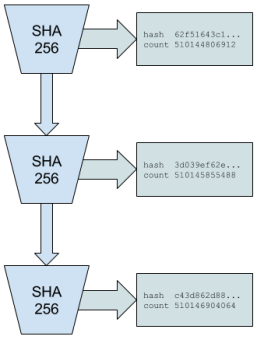


Figure 2: Proof of History sequence

is no way to predict what the hash value at index 300 is going to be without actually running the algorithm from the starting value 300 times. Thus we can infer from the data structure that time passed between index 0 and index 300.

In the example in Figure 2, hash `62f51643c1` was produced on count 510144806912 and hash `c43d862d88` was produced on count 510146904064. Real time passed between count 510144806912 and count 510146904064.

4.2 Timestamp for Events

This sequence of hashes can also be used to record that some piece of data was created before a particular hash index was generated. Using a 'combine' function to combine the piece of data with the current hash at the current index. The 'data' can simply be a cryptographically unique hash of arbitrary event data. The combine function can be a simple append of data, or any operation that is collision resistant. The next generated hash represents a "timestamp" of the data, because it could have only been generated after that specific piece of data was inserted.

For example:

PoH Sequence		
Index	Hash	Operation
1	hash1	sha256(”any random starting value”)
200	hash200	sha256(hash199)
300	hash300	sha256(hash299)

Some external event occurs, like a photograph was taken, or any arbitrary digital data was created:

PoH Sequence With Data		
Index	Hash	Operation
1	hash1	sha256(”any random starting value”)
200	hash200	sha256(hash199)
300	hash300	sha256(hash299)
336	hash336	sha256(append(hash335, photograph_sha256))

`Hash336` is computed from the appended binary data of `hash335` and the `sha256` of the photograph. The index, and the `sha256` of the photograph are recorded as part of the sequence output. So anyone verifying this sequence can then recreate this change to the sequence. The verifying can still be done in parallel and its discussed in Section 4.3

Because the initial process is still sequential, we can then tell that things entered into the sequence must have occurred sometime before the future hashed value was computed.

In the sequence represented by Table 1, `photograph2` was created before `hash600`, and `photograph1` was created before `hash336`. Inserting the data into the sequence of hashes results in a change to all subsequent values in the sequence. As long as the hash function used is collision resistant, and the data was appended, it is computationally impossible to pre-compute any future sequences based on prior knowledge of what data will be integrated

Index	Hash	Operation
1	hash1	sha256(”any random starting value”)
200	hash200	sha256(hash199)
300	hash300	sha256(hash299)
336	hash336	sha256(append(hash335, photograph1_sha256))
400	hash400	sha256(hash399)
500	hash500	sha256(hash499)
600	hash600	sha256(append(hash599, photograph2_sha256))
700	hash700	sha256(hash699)

Table 1: PoH Sequence With 2 Events

into the sequence.

The data that is mixed into the sequence can be the raw data itself, or just a hash of the data with accompanying metadata.

In the example in Figure 3, input `cfd40df8...` was inserted into the Proof of History sequence. The count at which it was inserted is 510145855488 and the state at which it was inserted it is `3d039eef3`. All the future generated hashes are modified by this change to the sequence, this change is indicated by the color change in the figure.

Every node observing this sequence can determine the order at which all events have been inserted and estimate the real time between the insertions.

4.3 Verification

The sequence can be verified as correct in a multi core computer in less time than it took to generated it

For example:

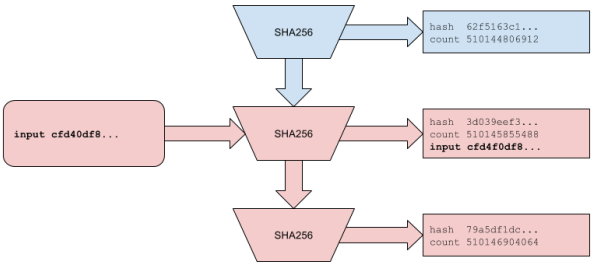


Figure 3: Figure description

Core 1		
Index	Hash	Data
200	hash200	sha256(hash199)
300	hash300	sha256(hash299)

Core 2		
Index	Hash	Data
300	hash300	sha256(hash299)
400	hash400	sha256(hash399)

So given some number of cores, like a modern GPU with 4000 cores, the verifier can split up the sequence of hashes and their indexes into 4000 slices, and in parallel make sure that each slice is correct from the starting hash to the last hash in the slice. So if the expected time to produce the sequence is going to be

$$\frac{\text{Total number of hashes}}{\text{Hashes per second for 1 core}}$$

The expected time to verify that the sequence is correct is going to be

$$\frac{\text{Total number of hashes}}{\text{(Hashes per second per core * Number of cores available to verify)}}$$

In the example in Figure 4, each core is able to verify each slice of the sequence in parallel. Since all input strings are recorded into the output, with the counter and state that they are appended to, the verifiers can replicate

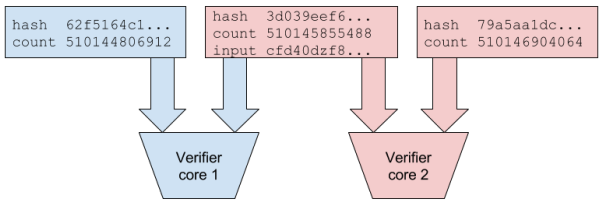


Figure 4: Verification using multiple cores

each slice in parallel. The red colored hashes indicate that the sequence was modified by a data insertion.

4.4 Horizontal Scaling

It's possible to synchronize multiple Proof of History generators by mixing the sequence state from each generator to each other generator, and thus achieve horizontal scaling of the Proof of History generator. This scaling is done without sharding, the output of both generators is necessary to reconstruct the full order of events in the system.

PoH Generator A		
Index	Hash	Data
1	hash1a	
2	hash2a	hash1b
3	hash3a	
4	hash4a	

PoH Generator B		
Index	Hash	Data
1	hash1b	
2	hash2b	hash1a
3	hash3b	
4	hash4b	

Given generators A and B, A receives a data packet from B (hash1b), which contains the last state from Generator B, and the last state generator B observed from Generator A. The next state hash in Generator A then depends on the state from Generator B, so we can derive that hash1b happened

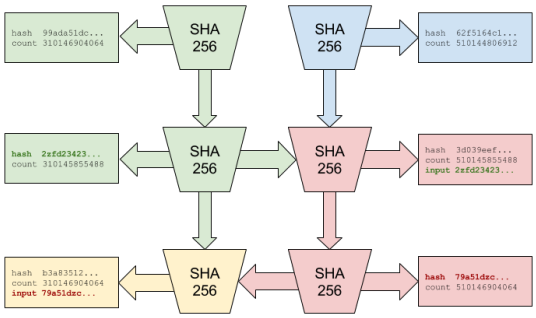


Figure 5: Two generators synchronizing

sometime before hash3a. This property can be transitive, so if three generators are synchronized through a single common generator $A \leftrightarrow B \leftrightarrow C$, we can trace the dependency between A and C even though they were not synchronized directly.

By periodically synchronizing the generators, each generator can then handle a portion of external traffic, thus the overall system can handle a larger amount of events to track at the cost of true time accuracy due to network latencies between the generators. A global order can still be achieved by picking some deterministic function to order any events that are within the synchronization window, such as by the value of the hash itself.

In Figure 5, the two generators insert each others output state and record the operation. The color change indicates that data from the peer had modified the sequence. The generated hashes that are mixed into each stream are highlighted in bold.

The synchronization is transitive. $A \leftrightarrow B \leftrightarrow C$ There is a provable order of events between A and C through B.

Scaling in this way comes at the cost of availability. 10x1gbps connections with availability of 0.999 would have $0.999^{10} = 0.99$ availability.

4.5 Consistency

Users can enforce consistency of the generated sequence and make it resistant to attacks by inserting the last observed output of the sequence they consider valid into their input.

PoH Sequence A		
Index	Hash	Data
10	hash10a	
20	hash20a	Event1
30	hash30a	Event2
40	hash40a	Event3

PoH Hidden Sequence B		
Index	Hash	Data
10	hash10a	
20	hash20a	Event3
30	hash30a	Event2
40	hash40a	Event1

A malicious PoH generator could produce a second hidden sequence with the events in reverse order, if it has access to all the events at once, or is able to generate a faster hidden sequence.

To prevent this attack, each client generated Event should contain within itself the latest hash that the client observed from what it considers to be a valid sequence. So when a client creates the "Event1" data, they should append the last hash they have observed.

PoH Sequence A		
Index	Hash	Data
10	hash10a	
20	hash20a	Event1 = append(event1 data, hash10a)
30	hash30a	Event2 = append(event2 data, hash20a)
40	hash40a	Event3 = append(event3 data, hash30a)

When the sequence is published, Event3 would be referencing hash30a, and if its not in the sequence prior to this Event, the consumers of the sequence know that its an invalid sequence. The partial reordering attack would then be limited to the number of hashes produced while the client has observed an event and when the event was entered. Clients can then write software that does not assume the order is correct for the short period of hashes between the last observed and inserted hash.

To prevent a malicious PoH generator from rewriting the client Event hashes, the clients can submit a signature of the event data and the last

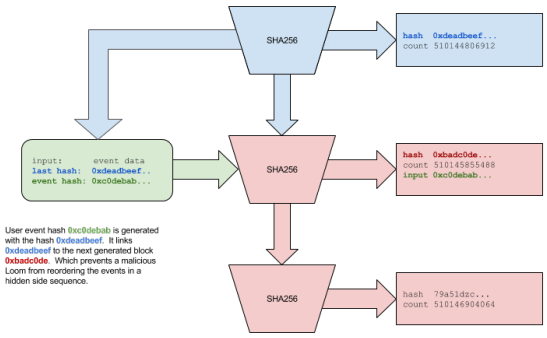


Figure 6: Input with a back reference.

observed hash instead of just the data.

PoH Sequence A		
Index	Hash	Data
10	hash10a	
20	hash20a	Event1 = sign(append(event1 data, hash10a), Client Private Key)
30	hash30a	Event2 = sign(append(event2 data, hash20a), Client Private Key)
40	hash40a	Event3 = sign(append(event3 data, hash30a), Client Private Key)

Verification of this data requires a signature verification, and a lookup of the hash in the sequence of hashes prior to this one.
Verify:

```
(Signature, PublicKey, hash30a, event3 data) = Event3
Verify(Signature, PublicKey, Event3)
Lookup(hash30a, PoHSequence)
```

In Figure 6, the user supplied input is dependent on hash 0xdeadbeef existing in the generated sequence sometime before its inserted. The blue top left arrow indicates that the client is referencing a previously produced hash. The clients message is only valid in a sequence that contains the hash

0xdeadbeef. The red color in the sequence indicates that the sequence has been modified by the clients data.

4.6 Overhead

4000 hashes per second would generate an additional 160 kilobytes of data, and would require access to a GPU with 4000 cores and roughly 0.75-0.25 milliseconds of time to verify.

4.7 Attacks

4.7.1 Reversal

Generating a reverse order would require an attacker to start the malicious sequence after the second event. This delay would allow any non malicious peer to peer nodes to communicate about the original order.

4.7.2 Speed

Having multiple generators may make deployment more resistant to attacks. One generator could be high bandwidth, and receive many events to mix into its sequence, another generator could be high speed low bandwidth that periodically mixes with the high bandwidth generator.

The high speed sequence would create a secondary sequence of data that an attacker would have to reverse.

4.7.3 Long Range Attacks

Long range attacks involve acquiring old discarded client Private Keys, and generating a falsified ledger [9]. Proof of history provides some protection against long range attacks. A malicious user that gains access to old private keys would have to recreate a historical record that takes as much time as the original one they are trying to forge. This would require access to a faster processor than the network is currently using, otherwise the attacker would never catch up in history length.

Additionally, a single source of time allows for construction of a simper Proof of Replication (more on that in Section 6). Since all the participants in the network can rely on a single historical record of events.

PoRep and PoH together provide a defense of both space and time against a forged ledger.

5 Proof of Stake Consensus

5.1 Description

This specific instance of Proof of Stake is designed for quick confirmation of the current sequence produced by the Proof of History generator, for voting and selecting the next Proof of History generator, and for punishing any misbehaving validators. This algorithm depends on messages eventually arriving to all participating nodes within a certain timeout.

5.2 Terminology

bonds Bonds are equivalent to a capital expense in Proof of Work. A miner buys hardware and electricity, and commits it to a single branch in a Proof of Work blockchain. A bond is coin that the validator commits as collateral while they are validating transactions.

slashing The solution to the nothing at stake problem in Proof of Stake systems [6]. When a proof of voting for a different branch is published, that branch can destroy the validator's bond. This is an economic incentive designed to discourage validators from confirming multiple branches.

super majority A super majority is $\frac{2}{3}$rds of the validators weighted by their bonds. A super majority vote indicates that the network has reached consensus, and at least $\frac{1}{3}$rd of the network would have had to vote maliciously for this branch to be invalid. This puts the economic cost of an attack at $\frac{1}{3}$rd of the market cap of the coin.

5.3 Bonding

A bonding transaction takes a user specified amount of coin and moves it to a bonding account under the users identity. Coins in the bonding account cannot be spent, and have to remain in the account until the user removes them. The user can only remove stale coins that have timed out. Bonds

are valid after super majority of the current stakeholders have confirmed the sequence.

5.4 Voting

Proof of History generator will publish a signature of the state at a predefined period. Each bonded identity must confirm that signature by publishing their own signed signature of the state. The vote is a simple Yes vote, without a no.

If super majority of the bonded identities have voted within a timeout, then this branch is accepted as valid.

5.5 Unbonding

Missing N number of votes marks the coins as stale and no longer eligible for voting. The user can issue an unbonding transaction to remove them.

N is a dynamic value based on the ratio of stale to active votes. N increases as the number of stale votes increase. In an event of a large network partition, this allows the larger branch to recover faster then the smaller branch.

5.6 Elections

Election for a new PoH generator occur when the PoH generator failure is detected. The validator with the largest voting power, or highest public key address if there is a tie is picked as the new PoH generator.

A super majority of confirmations are required on the new sequence. If the new leader fails before a super majority confirmations are available, the next highest validator is selected, and new set of conformations is required.

To switch votes, a validator needs to vote at a higher PoH sequence counter, and the new vote needs to contain the votes it wants to switch. Otherwise the second vote will be slashable. Vote switching can only occur at a height that does not have a super majority.

Once a PoH generator is established, a Secondary can be elected to take over the transactional processing duties. If a Secondary exists, it will be considered as the next leader during a Primary failure.

Secondary and lower rank generators are promoted to Primary at a predefined schedule, or if an exception is detected.

5.7 Election Triggers

5.7.1 Forked Proof of History generator

PoH generators have an identity that signs the generated sequence. A fork can only occur in case the PoH generator identity has been compromised. A fork is detected because two different historical records have been published on the same PoH identity.

5.7.2 Runtime Exceptions

A hardware failure or a bug, or a intentional error in the PoH generator could cause it to generate an invalid state and publish a signature of the state that does not match the local validators result. Validators will publish the correct signature via gossip and this event would trigger a new round of elections. Any validators who accept an invalid state will have their bonds slashed.

5.7.3 Network Timeouts

A network timeout would trigger an election.

5.8 Slashing

Slashing occurs when a validator votes two separate sequences. A proof of malicious vote will remove the bonded coins from circulation and add them to the mining pool.

A vote that includes a previous vote on a contending sequence is not eligible as proof of malicious voting. Instead of slashing the bonds, this vote removes remove the currently cast vote on the contending sequence.

Slashing also occurs if a vote is cast for an invalid hash generated by the PoH generator. The generator is expected to randomly generate an invalid state which would trigger a fallback to Secondary.

5.9 Secondary Elections

Secondary and lower ranked Proof of History generators can be proposed and approved. A proposal is cast on the primary generators sequence. The proposal contains a timeout, if the motion is approved by a super majority of the vote before the timeout, the Secondary is considered elected, and will

take over duties as scheduled. Primary can do a soft handover to Secondary by inserting a message into the generated sequence indicating that a handover will occur, or inserting an invalid state and forcing the network to fallback to Secondary.

If a Secondary is elected, and the primary fails, the Secondary will be considered as the first fallback during an election.

5.10 Attacks

5.10.1 Tragedy of Commons

The PoS verifiers simply confirm the state hash generated by the PoH generator. There is economic incentive for them to do no work and simply approve every generated state hash.

To avoid this condition, the PoH generator should generate an invalid hash with a probability P. Any voters for this hash should be slashed. When the hash is generated, the network should immediately promote the Secondary elected PoH generator.

5.10.2 Collusion with the PoH generator

A verifier that is colluding with the PoH generator would know in advance when the invalid hash is going to be produced and not vote for it. This scenario is really no different than the PoH identity having a larger verifier stake. The PoH generator still has to do all the work to produce the state hash.

5.10.3 Censorship

Censorship or denial of service could occur when $\frac{1}{3}$rd of the bond holders refuse to validate any sequences with new bonds. The protocol can defend against this form of attack by dynamically adjusting how fast bonds become stale. In the event of a denial of service, the larger partition can fork and censor the Byzantine bond holders. The larger network will recover as the Byzantine bonds become stale with time. The smaller Byzantine partition would not be able to move forward for a longer period of time.

The algorithm would work as follows. A majority of the network would elect a new Loom. The Loom would then censor the Byzantine bond holders

from participating. Proof of History generator would have to continue generating a sequence, to prove the passage of time, until enough Byzantine bonds have become stale so the bigger network has a super majority. The rate at which bonds become stale would be dynamically based on what percentage of bonds are active. So the Byzantine minority fork of the network would have to wait much longer than the majority fork to recover a super majority. Once a super majority has been established, slashing could be used to permanently punish the Byzantine bond holders.

6 Streaming Proof of Replication

6.1 Description

Filecoin proposed a version of Proof of Replication [5]. The goal of this version is to have fast and streaming verifications of Proof of Replication which are enabled by keeping track of time in Proof of History generated sequence. Replication is not used as a consensus algorithm, but is a useful tool to account for the cost of storing the blockchain history or state at a high availability.

6.2 Algorithm

As shown in Figure 7 CBC encryption encrypts each block of data in sequence, using the previously encrypted block to XOR the input data.

Each replication identity generates a key by signing a hash that has been generated Proof of History sequence. This ties the key to a replicators identity, and to a specific Proof of History sequence. Only specific hashes can be selected. (See Section 6.5 on Hash Selection)

The data set is fully encrypted block by block. Then to generate a proof, the key is used to seed a pseudorandom number generator that selects a random 32 bytes from each block.

A merkle hash is computed with the selected PoH hash prepended to the each slice.

The root is published, along with the key, and the selected hash that was generated. The replication node is required to publish another proof in N hashes as they are generated by Proof of History generator, where N is approximately $\frac{1}{2}$ the time it takes to encrypt the data. The Proof of

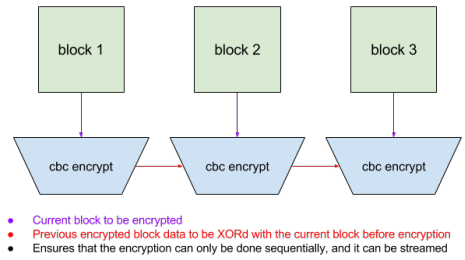


Figure 7: Sequential CBC encryption

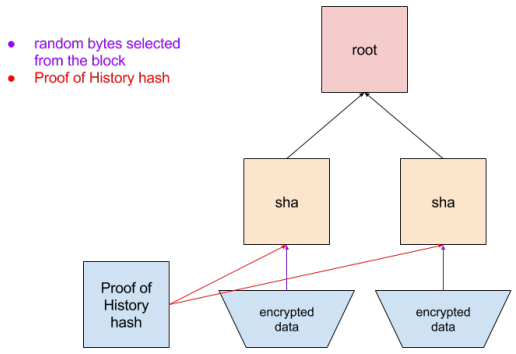


Figure 8: Fast Proof of Replication

History generator will publish specific hashes for Proof of Replication at a predefined periods. The replicator node must select the next published hash for generating the proof. Again, the hash is signed, and random slices are selected from the blocks to create the merkle root.

After a period of N proofs, the data is re-encrypted with a new CBC key.

6.3 Verification

With N cores, each core can stream encryption for each identity. Total space required is $2blocks * Ncores$, since the previous encrypted block is necessary to generate the next one. Each core can then be used to generate all the proofs that derived from the current encrypted block.

Total time to verify proofs is equal to the time it takes to encrypt. The proofs themselves consume few random bytes from the block, so the amount of data to hash is significantly lower then the encrypted block size. The number of replication identities that can be verified at the same time is equal to the number of available cores. Modern GPUs have 3500+ cores available to them, albeit at $\frac{1}{2}$-$\frac{1}{3}$rd the clock speed of a CPU.

6.4 Key Rotation

Without key rotation the same encrypted replication can generate cheap proofs for multiple Proof of History sequences. Keys are rotated periodically and each replication is re-encrypted with a new key that is tied to a unique Proof of History sequence.

Rotation needs to be slow enough that its practical to verify replication proofs on GPU hardware, which is slower per core than CPUs.

6.5 Hash Selection

Proof of History generator publishes a hash to be used by the entire network for encrypting Proofs of Replication, and for using as the pseudorandom number generator for byte selection in fast proofs.

Hash is published at a periodic counter that is roughly equal to $\frac{1}{2}$ the time it takes to encrypt the data set. Each replication identity must use the same hash, and use the signed result of the hash as the seed for byte selection, or the encryption key.

The period that each replicator must provide a proof must be smaller then the encryption time. Otherwise the replicator can stream the encryption and delete it for each proof.

A malicious generator could inject data into the sequence prior to this hash to generate a specific hash. This attack is discussed more in 5.10.2.

6.6 Proof Validation

The Proof of History node does not validate the submitted Proof of Replication proofs. It only keep track of number of pending and verified proofs submitted by the replicators identity. A proof becomes verified when the replicator is able to sign the proof by a super majority of the validators in the network.

The verifications are collected by the replicator via p2p gossip network, and submitted as one packet that contains a super majority of the validators in the network. This packet verifies all the proofs prior to a specific hash generated by the Proof of History sequence, and can contain multiple replicator identities at once.

6.7 Attacks

6.7.1 Spam

A malicious user could create many replicator identities and spam the network with bad proofs. To facilitate faster verification, nodes are required to provide the encrypted data and the entire merkle tree to the rest of the network when they request verification.

The Proof of Replication that is designed in this paper allows for cheap verification of any additional proofs, as they take no additional space. But each identity would consume 1 core of encryption time. The replication target should be set to a maximum size of readily available cores. Modern GPUs ship with 3500+ cores.

6.7.2 Partial Erasure

A replicator node could attempt to partially erase some of the data to avoid storing the entire state. The number of proofs, and the randomness of the seed should make this attack difficult.

For example, a user storing 1 terabyte of data erases a single byte from each 1 megabyte block. A single proof that samples 1 byte out of every megabyte would have a likelihood of collision with any erased byte $1 - (1 - 1/1,000,0000)^{1,000,000} = 0.63$. After 5 proofs the likelihood is 0.99.

6.7.3 Collusion with PoH generator

The signed hash is used to seed the sample. If a replicator could select a specific hash in advance then the replicator could erase all bytes that are not going to be sampled.

A replicator identity that is colluding with the proof of history generator could inject a specific transaction at the end of the sequence before the predefined hash for random byte selection is generated. With enough cores, an attacker could generate a hash that is preferable to the replicator identity.

This attack could only benefit a single replicator identity. Since all the identities have to use the same exact hash that is cryptographically signed with ECDSA (or equivalent), the resulting signature is unique for each replicator identity, and collision resistant. A single replicator identity would only have marginal gains.

6.7.4 Denial of Service

The cost of adding an additional replicator identity is equal to the cost of storage. The cost of adding extra computational capacity to verify all the replicator identities is equal to the cost of a CPU or GPU core per replication identity.

This creates an opportunity for a denial of service attack on the network by creating a large number of valid replicator identities.

To limit this attack, the consensus protocol chosen for the network can select a replication target, and award the replication proofs that meet the desired characteristics, like availability on the network, bandwidth, geolocation etc...

6.7.5 Tragedy of Commons

The PoS verifiers could simply confirm PoRep without doing any work. The economic incentives should be lined up with the PoS verifiers to do work, like by splitting the mining payout between the PoS verifiers and the PoRep replication nodes.

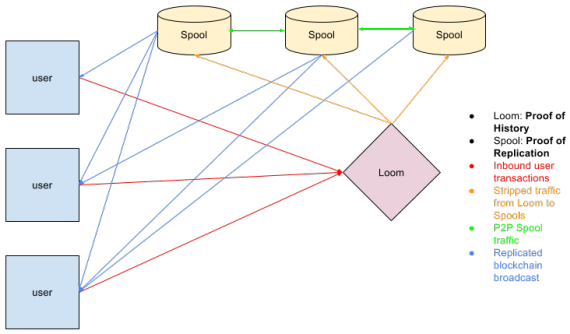


Figure 9: System Architecture

To further avoid this scenario, the PoRep verifiers can submit false proofs a small percentage of the time. They can prove the proof is false by providing the function that generated the false data. Any PoS verifier that confirmed a false proof would be slashed.

7 System Architecture

7.1 Components

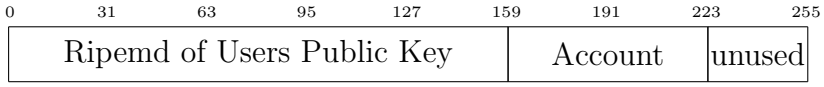
7.1.1 Loom, Proof of History generator

The Loom is an elected Proof of History generator. It consumes arbitrary user transactions and outputs a Proof of History sequence of all the transactions that guarantees a unique global order in the system. After each batch of transactions the Loom outputs a signature of the state that is the result of running the transactions in that order. This signature is signed with the identity of the Loom.

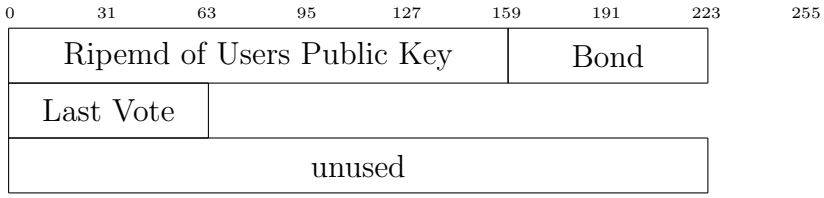
7.1.2 State

A naive hash table indexed by the users address. Each cell contains the full users address and the memory required for this computation. For example

Transaction table contains:



For a total of 32 bytes.
Proof of Stake bonds table contains:



For a total of 64 bytes.

7.1.3 Spool, State Replication

The Spool nodes replicate the blockchain state and provide high availability of the blockchain state. The replication target is selected by the consensus algorithm, and the validators in the consensus algorithm select and vote the Proof of Replication nodes they approve of based on of-chain defined criteria.

The network could be configured with a minimum Proof of Stake bond size, and a requirement for a single replicator identity per bond.

7.1.4 Validators

These nodes are consuming bandwidth from Spools. They are virtual nodes, and can run on the same machines as the Spools or the Loom, or on separate machines that are specific to the consensus algorithm configured for this network.

7.2 Network Limits

Loom takes incoming user packets, orders them the most efficient way possible, and sequences them into a Proof of History sequence that is published to downstream Spools. Efficiency is based on memory access patterns of the transactions, so the transactions are ordered to minimize faults and to maximize prefetching.

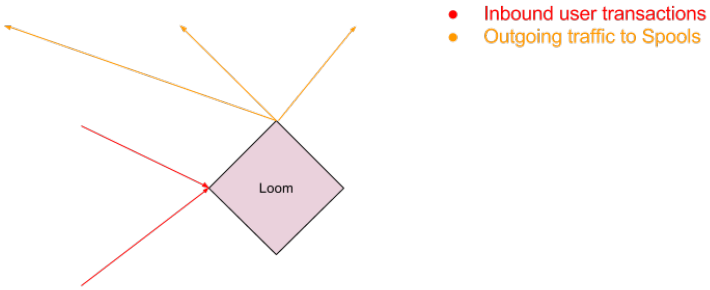
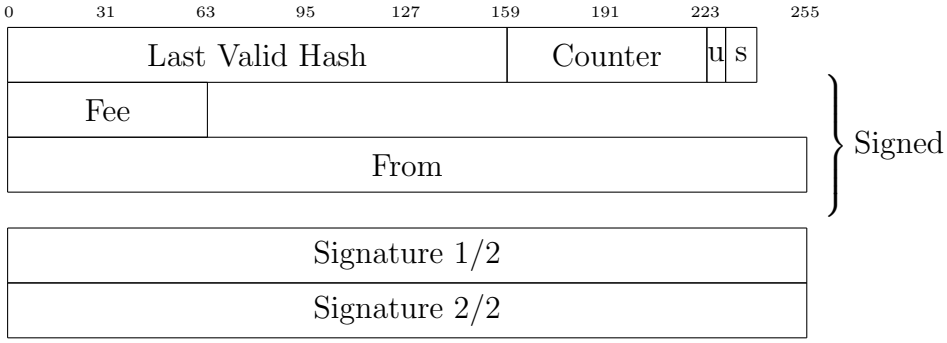


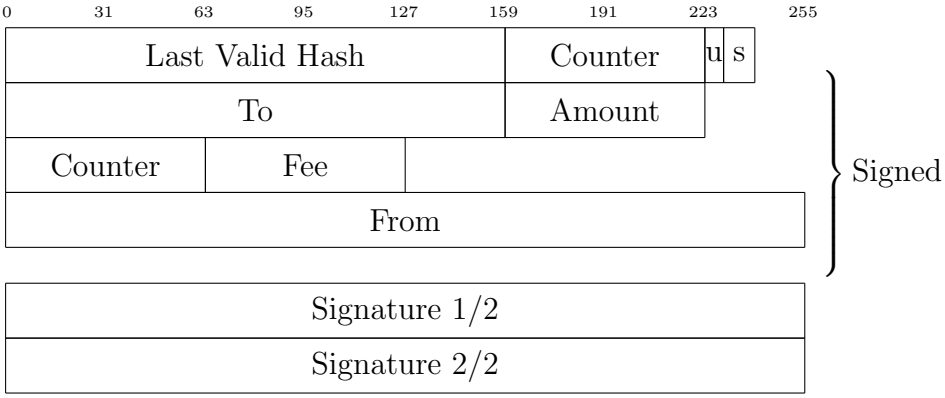
Figure 10: Loom network limits

Incoming packet format:



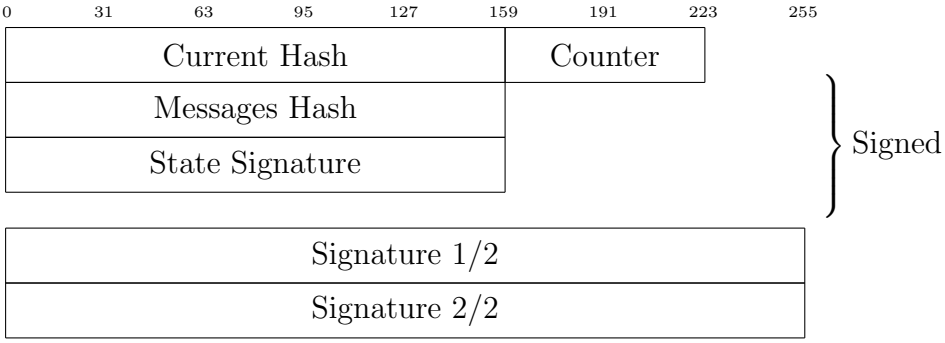
Size $20 + 8 + 16 + 8 + 32 + 3232 = 148$ bytes.

The minimal payload that can be supported would be 1 destination account. With Payload:



With payload the minimum size: 176 bytes

The Proof of History sequence packet contains the current hash, counter, and the hash of all the new messages added to the PoH sequence and the state signature after processing all the messages. This packet is sent once every N messages are broadcast.
Proof of History packet:



Minimum size of the output packet is: 132 bytes

On a 1gbps network connection the maximum number of transactions possible is 1 gigabit per second / 176 bytes = 710k tps max. Some loss $1-4\%$ is expected due to Ethernet framing. The spare capacity over the target amount for the network can be used to increase availability by coding the output with Reed-Solomon codes and striping it to the available downstream

Spools.

7.3 Computational Limits

Each transaction requires a digest verification. This operation does not use any memory outside of the transaction message itself, and can be parallelized independent of all the other transactions. Thus throughput is going to be limited by the number of cores available on the system.

GPU based ECDSA verification servers have had experimental results of 900k operations per second [8].

7.4 Memory Limits

A naive implementation of the state as a 50% full hashtable with 32 byte entries for each account, would fit 10 billion accounts into 640GB. Steady state random access to this table is measured at $1.1 * 10^7$ writes or reads per second. Based on 2 reads and two writes per transaction, memory throughput can handle 2.75m transactions per second. This was measured on AWS 1TB x1.16xlarge instance.

7.5 High Performance Smart Contracts

Smart contracts are a generalized form of transactions. These are programs that run on each node and modify the state. This design leverages extended Berkeley Packet Filter bytecode as fast and easy to analyze and JIT bytecode as the smart contracts language.

One of its main advantages is a zero cost Foreign Function Interface. Intrinsics, or high level functions that are implemented on the platform directly, are callable by user supplied programs. Calling the intrinsics suspends that program and schedules the intrinsic on a high performance server. Intrinsics are batched together to execute in parallel on the GPU.

In the above example, two different user programs call the same intrinsic. Each program is suspended until the batch execution of the intrinsics is complete. An example common intrinsic is ECDSA verification. Batching these calls to execute on the GPU can increase throughput by thousands of times.

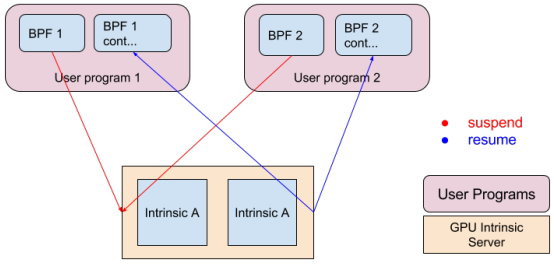


Figure 11: Executing user supplied BPF programs.

This trampoline requires no native Operating System thread context switches, since the BPF bytecode has a well defined context for all the memory that is using.

eBPF backend has been included in LLVM since 2015, so any LLVM frontend language can be used to write smart contracts. Its been in the Linux kernel since 2015, and the first iterations of the bytecode have been around since 1992. A single pass can check eBPF for correctness, ascertain its runtime and memory requirements and convert it to x86 instructions.

References

[1] Liskov, Practical use of Clocks
`http://www.dainf.cefetpr.br/ tacla/SDII/PracticalUseOfClocks.pdf`

[2] Google Spanner TrueTime consistency
`https://cloud.google.com/spanner/docs/true-time-external-consistency`

[3] Solving Agreement with Ordering Oracles
`http://www.inf.usi.ch/faculty/pedone/Paper/2002/2002EDCCb.pdf`

[4] Tendermint: Consensus without Mining
`https://tendermint.com/static/docs/tendermint.pdf`

[5] Filecoin, proof of replication,
https://filecoin.io/proof-of-replication.pdf

[6] Slasher, A punative Proof of Stake algorithm
https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorith

[7] BitShares Delegated Proof of Stake
https://github.com/BitShares/bitshares/wiki/Delegated-Proof-of-Stake

[8] An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration
http://ieeexplore.ieee.org/document/7555336/

[9] Casper the Friendly Finality Gadget
https://arxiv.org/pdf/1710.09437.pdf