Solana: A new architecture for a high performance blockchain v0.8.14

Anatoly Yakovenko
anatoly@solana.io

Abstract

This paper proposes a new blockchain architecture based on Proof of History (PoH) - a proof for verifying order and passage of time between events. PoH is used to encode trustless 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, resulting in sub-second finality times. This paper also proposes two algorithms that leverage the time keeping properties of the PoH ledger - a PoS 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 (ordering) and storage. The protocol is analyzed on a 1 gbps 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, 5]. 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 is designed to create a ledger with verifiable passage of time, i.e. duration between events and message ordering. It is anticipated that every node in the network will be able to 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 Network Design

As shown in Figure 1, at any given time a system node is designated as Leader to generate a Proof of History sequence, providing the network global read consistency and a verifiable passage of time. The Leader sequences user messages and orders them such that they can be efficiently processed by other nodes in the system, maximizing throughput. It executes the transactions on the current state that is stored in RAM and publishes the transactions and a signature of the final state to the replications nodes called Verifiers. Verifiers 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.

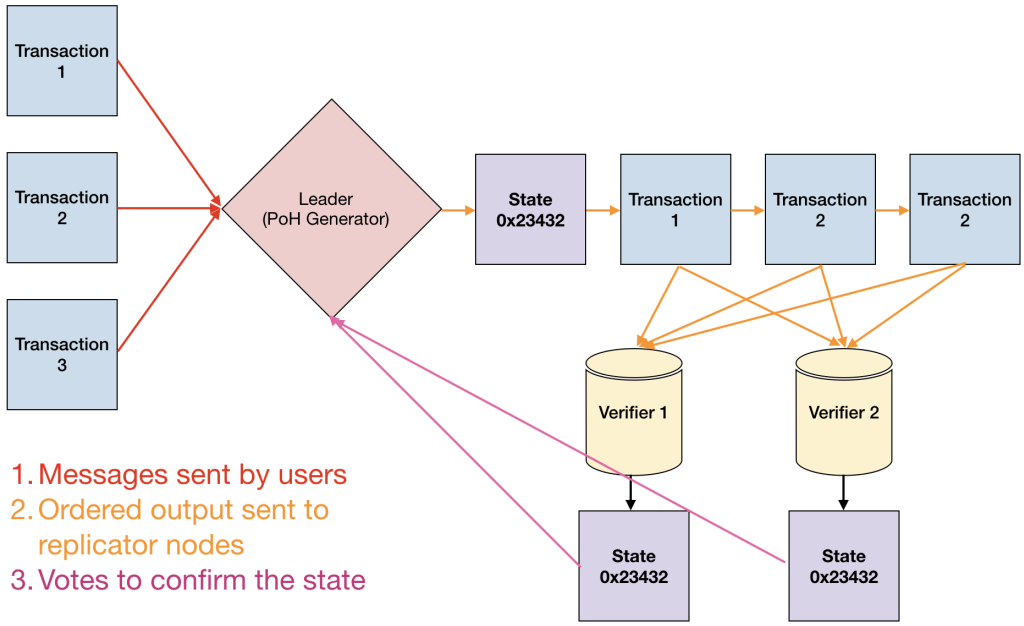


Figure 1: Transaction flow throughout the network.

In a non-partitioned state, at any given time, there is one Leader in the network. Each Verifier node has the same hardware capabilities as a Leader and can be elected as a Leader, this is done through PoS based elections. Elections for the proposed PoS algorithm are covered in depth in Section 5.6.

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 recover control of the network from a partition of any size. This is covered in depth in Section 5.12.

4 Proof of History

Proof of History is a sequence of computation that can provide a way to cryptographically verify passage of time between two events. It uses a cryptographically secure function written so that output cannot be predicted from the input, and must be completely executed to generate the output. The function is run in a sequence on a single core, 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 sequence segment on a separate core.

Data can be timestamped into this sequence by appending the data (or a hash of some 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 can guarantee that the data was created sometime before the next hash was generated in the sequence. This design also supports horizontal scaling 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

The system is designed to work as follows. With a cryptographic hash function, whose output cannot be predicted without running the function (e.g. `sha256`, `ripemd`, etc.), 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	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
2	sha256(`hash1`)	`hash2`
3	sha256(`hash2`)	`hash3`

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	Operation	Output Hash
1	sha256(”any random starting value”)	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

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. This follows from the fact that there 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 thus infer from the data structure that real time has 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. Following the previously discussed properties of the PoH algorithm, we can trust that 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	Operation	Output Hash
1	sha256(”any random starting value”)	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

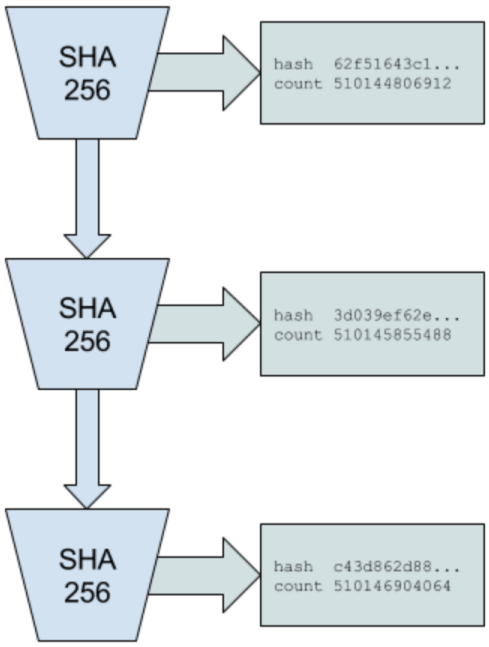


Figure 2: Proof of History sequence

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

PoH Sequence With Data

Index	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`
336	sha256(append(`hash335`, photograph_sha256))	`hash336`

`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 it's discussed in Section 4.3

POH Sequence

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

Table 1: PoH Sequence With 2 Events

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 should be computationally impossible to pre-compute any future sequences based on prior knowledge of what data will be integrated 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 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.

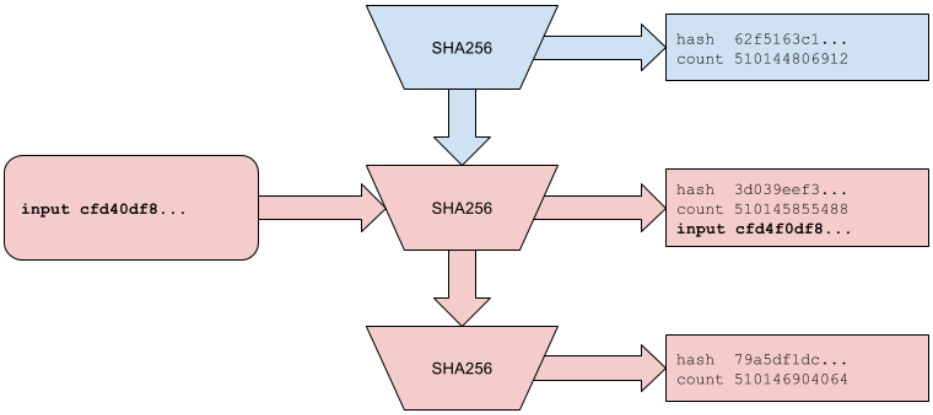


Figure 3: Inserting data into Proof of History

4.3 Verification

The sequence can be verified correct by a multicore computer in significantly less time than it took to generate it.

For example:

Core 1		
Index	Data	Output Hash
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`
Core 2		
Index	Data	Output Hash
300	sha256(`hash299`)	`hash300`
400	sha256(`hash399`)	`hash400`

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. If the expected time to produce the sequence is going to be:

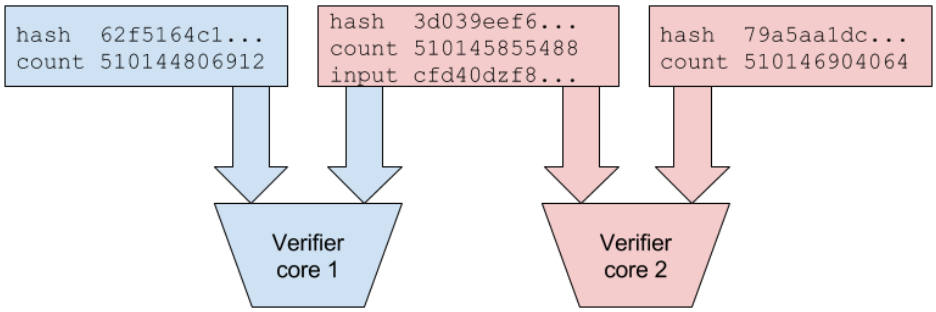


Figure 4: Verification using multiple cores

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

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 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 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 other's 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. 10×1 gbps connections with availability of 0.999 would have $0.999^{10} = 0.99$ availability.

4.5 Consistency

Users are expected to be able to 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.

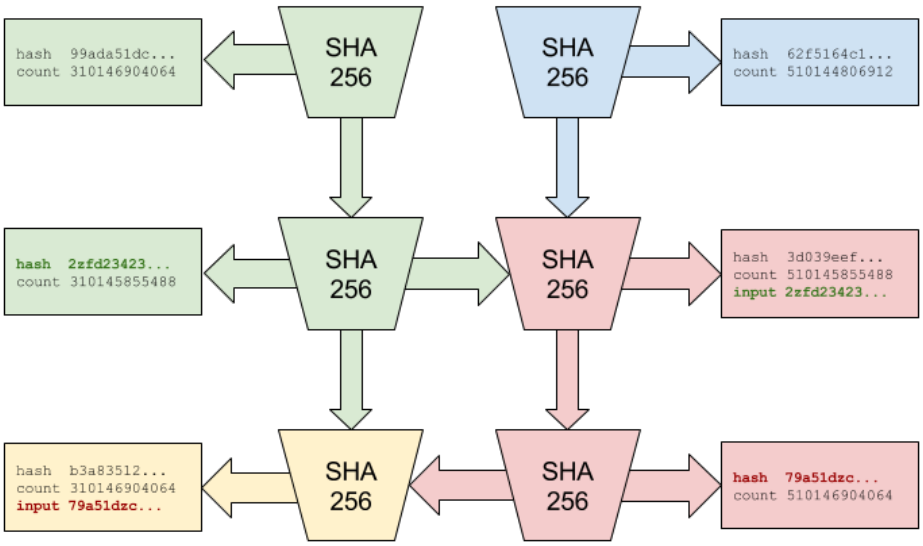


Figure 5: Two generators synchronizing

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

PoH Hidden Sequence B		
Index	Data	Output Hash
10		`hash10b`
20	Event3	`hash20b`
30	Event2	`hash30b`
40	Event1	`hash40b`

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	Data	Output Hash
10		`hash10a`
20	Event1 = append(event1 data, `hash10a`)	`hash20a`
30	Event2 = append(event2 data, `hash20a`)	`hash30a`
40	Event3 = append(event3 data, `hash30a`)	`hash40a`

When the sequence is published, Event3 would be referencing hash30a, and if it's not in the sequence prior to this Event, the consumers of the sequence know that it's 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 should then be able to 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 observed hash instead of just the data.

PoH Sequence A

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

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 it's inserted. The blue

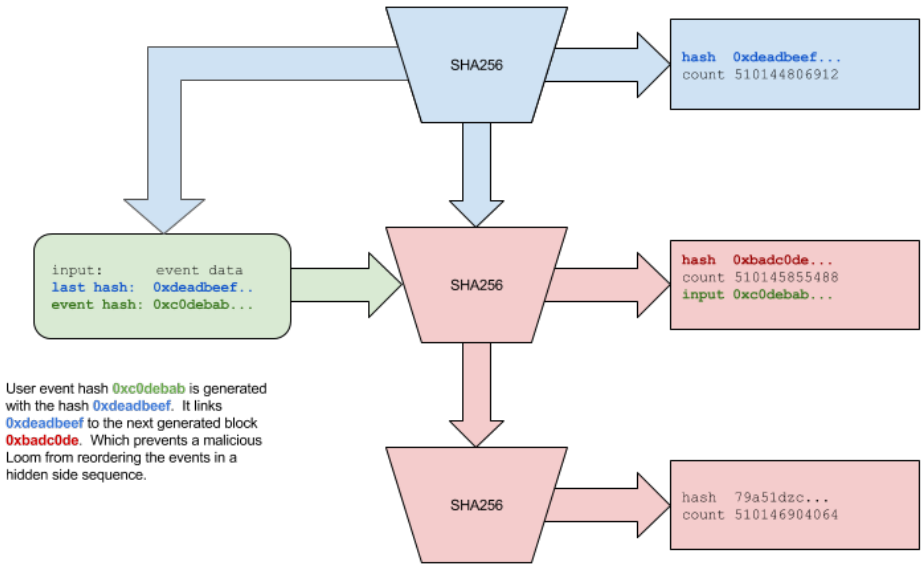


Figure 6: Input with a back reference.

top left arrow indicates that the client is referencing a previously produced hash. The client's 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 0.25-0.75 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 should 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 [10]. 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 simpler Proof of Replication (more on that in Section 6). Since the network is designed so that all participants in the network will rely on a single historical record of events.

PoRep and PoH together should 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 proposed solution to the nothing at stake problem in Proof of Stake systems [7]. 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 would put 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 amount of coin and moves it to a bonding account under the user's 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

It is anticipated that the Proof of History generator will be able to 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 would be 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 increases. In an event of a large network partition, this allows the larger branch to recover faster than 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 a new set of confirmations 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 is expected to be designed so that it 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.

The platform is designed so that the Secondary becomes Primary and lower rank generators are promoted if an exception is detected or on a predefined schedule.

5.7 Election Triggers

5.7.1 Forked Proof of History generator

PoH generators are designed with an identity that signs the generated sequence. A fork can only occur in case the PoH generator's 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 validator's 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 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 generator's 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 Availability

CAP systems that deal with partitions have to pick Consistency or Availability. Our approach eventually picks Availability, but because we have an objective measure of time, Consistency is picked with reasonable human timeouts.

Proof of Stake verifiers lock up some amount of coin in a "stake", which allows them to vote for a particular set of transactions. Locking up coin is a transaction that is entered into a PoH stream, just like any other transaction. To vote, a PoS verifier has to sign the hash of the state, as it was computed after processing all the transactions to a specific position in the PoH ledger. This vote is also entered as a transaction into the PoH stream. Looking at the PoH ledger, we can then infer how much time passed between each vote, and if a partition occurs, for how long each verifier has been unavailable.

To deal with partitions with reasonable human timeframes, we propose a dynamic approach to "unstake" unavailable verifiers. When the number of verifiers is high and above $\frac{2}{3}$, the "unstaking" process can be fast. The number of hashes that must be generated into the ledger is low before the unavailable verifiers stake is fully unstaked and they are no longer counted for consensus. When the number of verifiers is below $\frac{2}{3}$rds but above $\frac{1}{2}$, the unstaking timer is slower, requiring a larger number of hashes to be generated before the missing verifiers are unstaked. In a large partition, like a partition that is missing $\frac{1}{2}$ or more of the verifiers, the unstaking process is very very slow. Transactions can still be entered into the stream, and verifiers can still vote, but full $\frac{2}{3}$rds consensus will not be achieved until a very large amount of hashes have been generated and the unavailable verifiers have been unstaked. The difference in time for a network to regain liveness allows us as customers of the network human timeframes to pick a partition that we want to continue using.

5.11 Recovery

In the system we propose, the ledger can be fully recovered from any failure. That means, anyone in the world can pick any random spot in the ledger and create a valid fork by appending newly generated hashes and transactions. If all the verifiers are missing from this fork, it would take a very very long time for any additional bonds to become valid and for this branch to achieve $\frac{2}{3}$rds super majority consensus. So full recovery with zero available validators

would require a very large amount of hashes to be appended to the ledger, and only after all the unavailable validators have been unstaked will any new bonds be able to validate the ledger.

5.12 Finality

PoH allows verifiers of the network to observe what happened in the past with some degree of certainty of the time of those events. As the PoH generator is producing a stream of messages, all the verifiers are required to submit their signatures of the state within 500ms. This number can be reduced further depending on network conditions. Since each verification is entered into the stream, everyone in the network can validate that every verifier submitted their votes within the required timeout without actually observing the voting directly.

5.13 Attacks

5.13.1 Tragedy of Commons

The PoS verifiers simply confirm the state hash generated by the PoH generator. There is an economic incentive for them to do no work and simply approve every generated state hash. To avoid this condition, the PoH generator should inject an invalid hash at a random interval. Any voters for this hash should be slashed. When the hash is generated, the network should immediately promote the Secondary elected PoH generator.

Each verifier is required to respond within a small timeout - 500ms for example. The timeout should be set low enough that a malicious verifier has a low probability of observing another verifiers vote and getting their votes into the stream fast enough.

5.13.2 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.

5.13.3 Censorship

Censorship or denial of service could occur when a $\frac{1}{3}$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 will be designed to 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 Leader. The Leader 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.

5.13.4 Long Range Attacks

PoH provides a natural defense against long range attacks. Recovering the ledger from any point in the past would require the attacker to overtake the valid ledger in time by outpacing the speed of the PoH generator.

The consensus protocol provides a second layer of defense, as any attack would have to take longer than the time it takes to unstake all the valid validators. It also creates an availability "gap" in the history of the ledger. When comparing two ledgers of the same height, the one with the smallest maximum partition can be objectively considered as valid.

5.13.5 ASIC Attacks

Two opportunities for ASIC attacks exist in this protocol - during partition, and cheating timeouts in Finality.

For ASIC attacks during Partitions, the Rate at which bonds are unstaked is non-linear, and for networks with large partitions the rate is orders of magnitude slower than expected gains from an ASIC attack.

For ASIC attacks during Finality, the vulnerability allows for byzantine validators who have a bonded stake to wait for confirmations from other nodes and inject their votes with a collaborating PoH generator. The PoH generator can then use its faster ASIC to generate 500ms worth of hashes in less time, and allow for network communication between PoH generator and the collaborating nodes. But, if the PoH generator is also byzantine, there is no reason why the byzantine generator wouldn't have communicated the exact counter when they expect to insert the failure. This scenario is no different than a PoH generator and all the collaborators sharing the same identity vs. having a single combined stake and only using 1 set of hardware.

6 Streaming Proof of Replication

6.1 Description

Filecoin proposed a version of Proof of Replication [6]. 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 a 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 via a Proof of History sequence. This ties the key to a replicator's 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

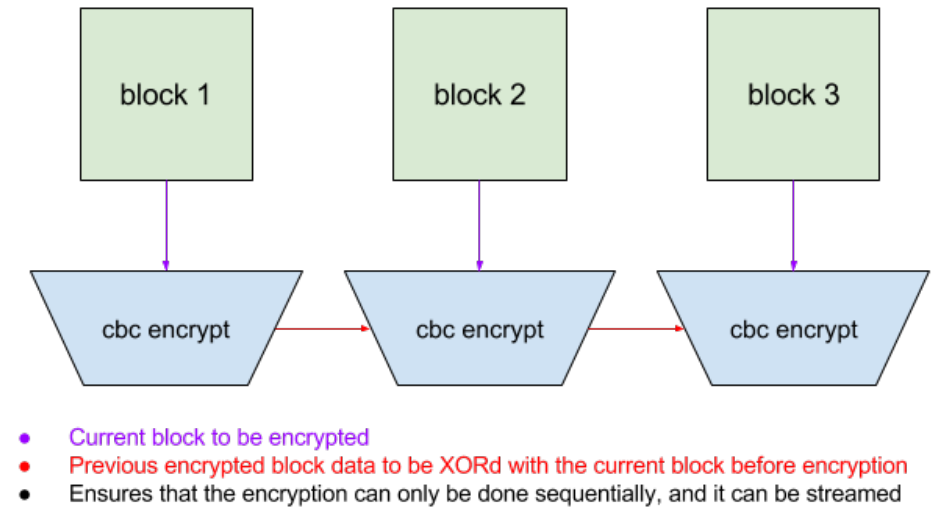


Figure 7: Sequential CBC encryption

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 History generator will publish specific hashes for Proof of Replication at 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 were derived from the current encrypted block.

Total time to verify proofs is expected to be 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 than 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 $\frac{1}{2}$-$\frac{1}{3}$ the clock speed of a CPU.

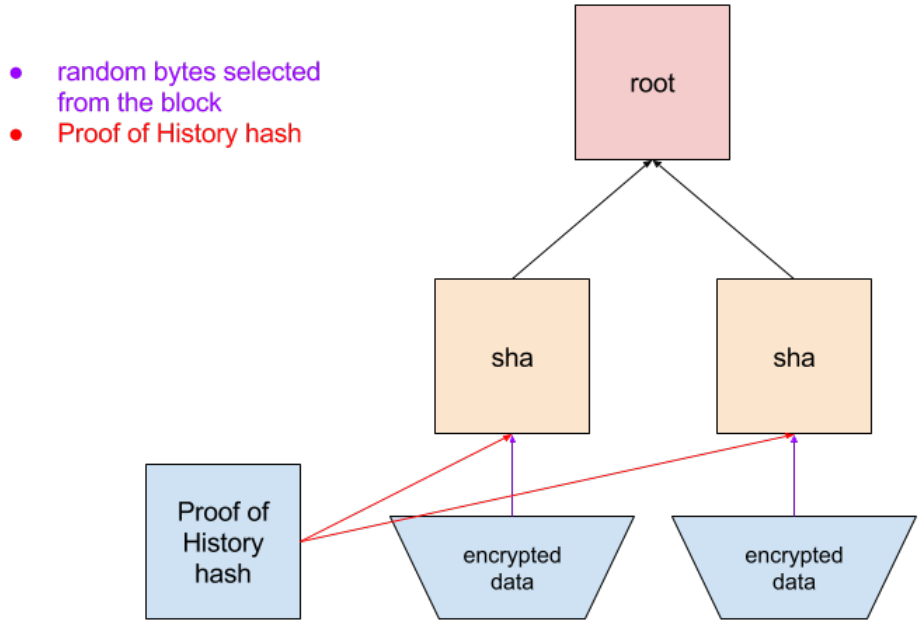


Figure 8: Fast Proof of Replication

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 it's practical to verify replication proofs on GPU hardware, which is slower per core than CPUs.

6.5 Hash Selection

The 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.

The 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 than 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 5.13.2.

6.6 Proof Validation

The Proof of History node is not expected to validate the submitted Proof of Replication proofs. It is expected to keep track of the number of pending and verified proofs submitted by the replicator's identity. A proof is expected to be 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 expected to be 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's 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 expected to be equal to the cost of storage. The cost of adding extra computational capacity to verify all the replicator identities is expected to be 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, i.e. splitting the mining payout between the PoS verifiers and the PoRep replication nodes.

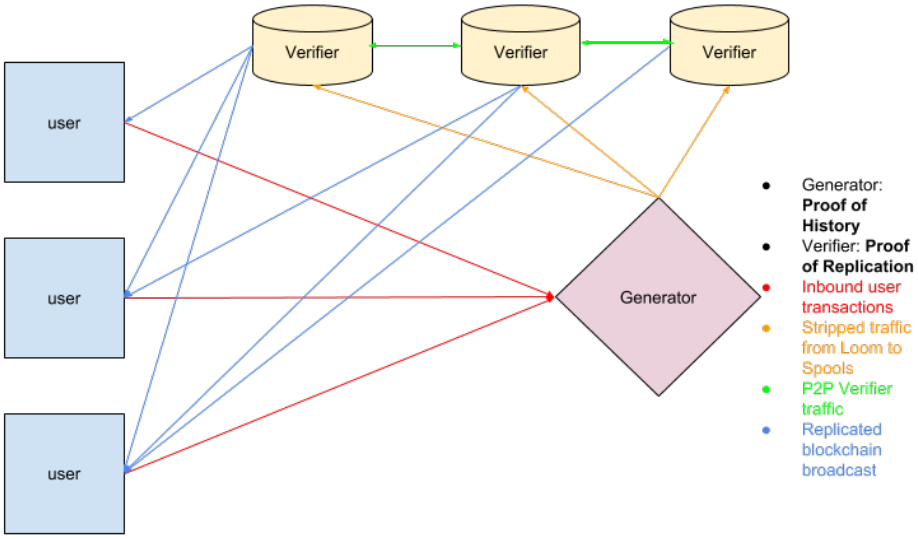


Figure 9: System Architecture

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

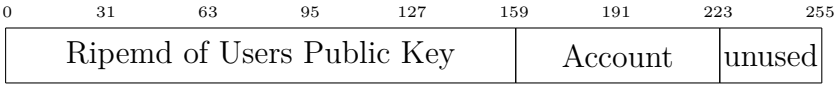
7.1 Components

7.1.1 Leader, Proof of History generator

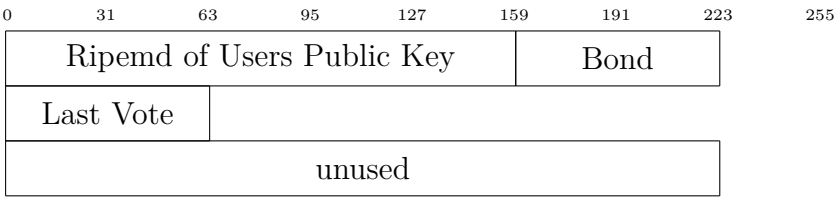
The Leader is an elected Proof of History generator. It consumes arbitrary user transactions and outputs a Proof of History sequence of all the transactions that guarantee a unique global order in the system. After each batch of transactions the Leader outputs a signature of the state that is the result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

7.1.2 State

The state is a naive hash table indexed by the user's address. Each cell contains the full user's address and the memory required for this computation. For example,
the Transaction table contains:



for a total of 32 bytes.
The Proof of Stake bond's table contains:



for a total of 64 bytes.

7.1.3 Verifier, State Replication

The Verifier nodes replicate 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 off-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 Verifiers. They are virtual nodes, and can run on the same machines as the Verifiers or the Leader, or on separate machines that are specific to the consensus algorithm configured for this network.

7.2 Network Limits

The leader is expected to be able to take incoming user packets, order them in the most efficient way possible, and sequence them into a Proof of History

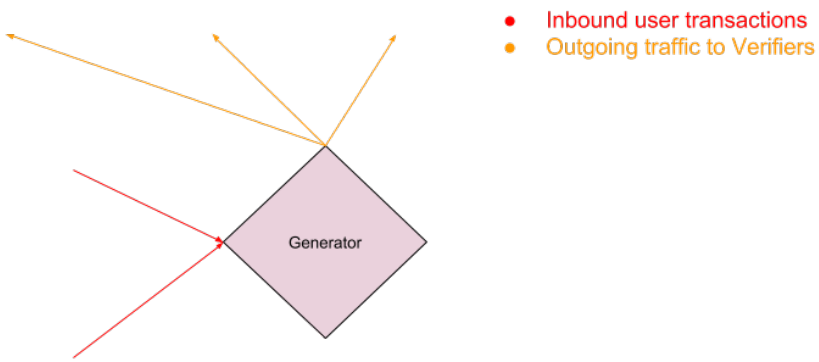
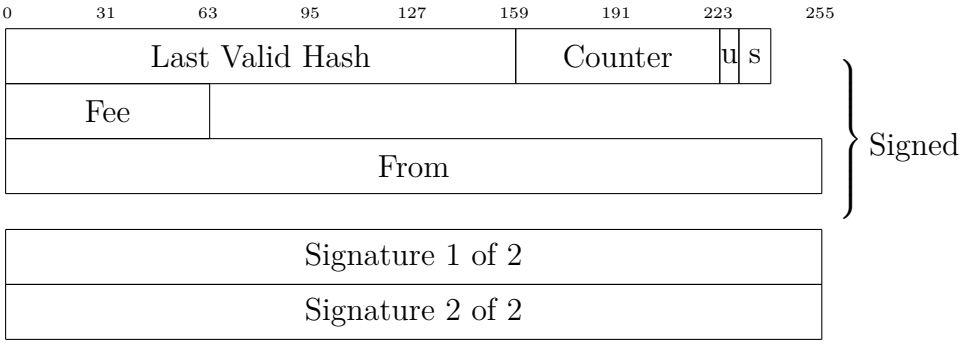


Figure 10: Generator network limits

sequence that is published to downstream Verifiers. Efficiency is based on memory access patterns of the transactions, so the transactions are ordered to minimize faults and to maximize prefetching.

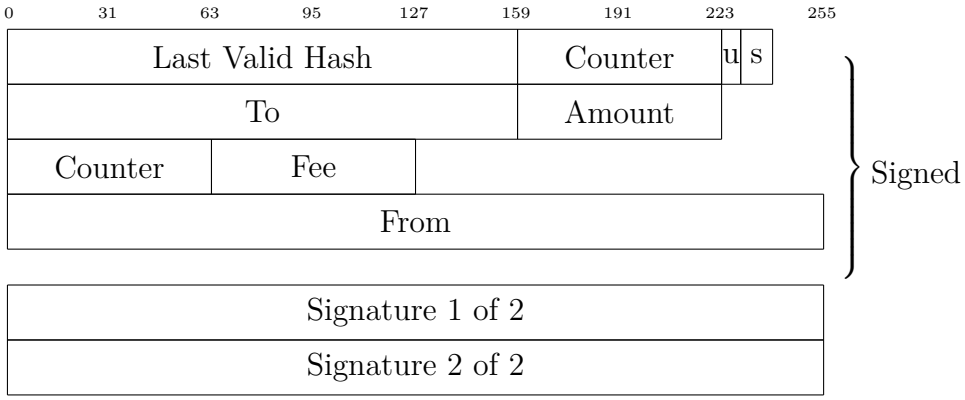
Incoming packet format:



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

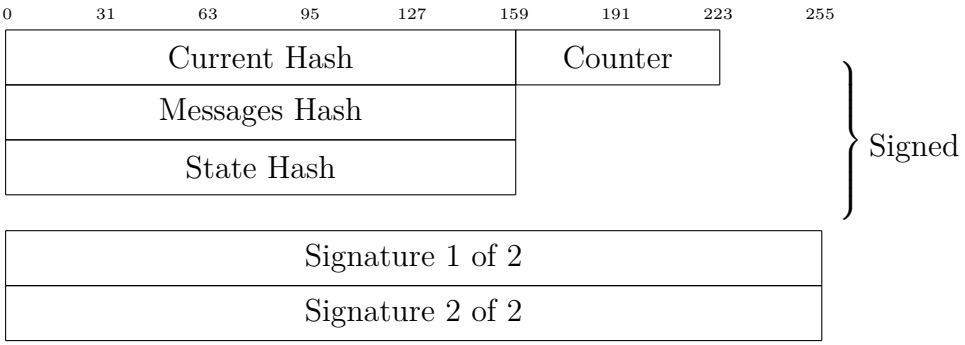
The minimal payload that can be supported would be 1 destination account with a minimum size of 176 bytes.

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 Verifiers.

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 independently. Thus throughput is expected 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 [9].

7.4 Memory Limits

A naive implementation of the state as a 50% full hashtable with 32 byte entries for each account, would theoretically 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 an Amazon Web Services 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, which is 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 functions that are implemented on the platform directly, are callable by 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 intrinsic is ECDSA verification. Batching these calls to execute on the GPU can increase throughput by thousands of times.

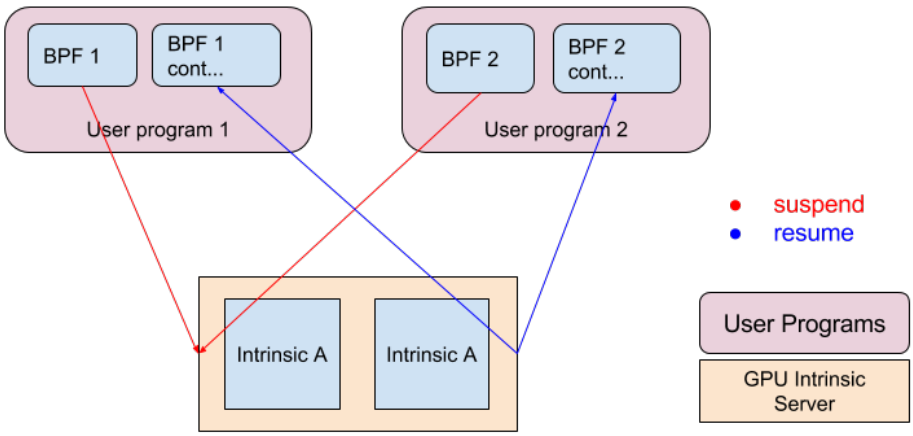


Figure 11: Executing BPF programs.

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

eBPF backend has been included in LLVM since 2015, so any LLVM frontend language can be used to write smart contracts. It's 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] Hedera: A Governing Council & Public Hashgraph Network
https://s3.amazonaws.com/hedera-hashgraph/hh-whitepaper-v1.0-180313.pdf

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

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

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

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

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