RENEC Blockchain - The new standard for Web3 Infrastructure

RENEC Foundation

2022

Abstract

The rise of blockchains era as the new Internet infrastructure has led to developers and content creators deploying, creating tens of thousands of decentralized applications and crypto assets at rapidly growing rates.

Unfortunately, blockchain usage is not yet universal due to frequent outages, high costs, low throughput limits, and numerous security concerns. To enable mass adoption in the web3 era, blockchain infrastructure needs to follow the path of cloud infrastructure as a trusted, scalable, cost-efficient, and continually improving platform for building widely-used applications.

In the web2 version of the Internet, services such as messaging, social media, finance, gaming, shopping, and audio/video streaming, are provided by centralized companies that control direct access to user data (e.g., Google, Amazon, Apple, and Meta). These companies develop infrastructure using application-specific software optimized for targeted use cases and leverage cloud infrastructures to deploy these applications to users. Cloud infrastructure provides access to virtualized and/or physical infrastructure services, such as rented virtual machines (VMs) and bare metal hardware operating inside data centers worldwide (e.g., AWS, Azure, and Google Cloud). As a result, building web2 Internet services that can scale to billions of users has never been easier than it is today. However, web2 requires that users place explicit trust in centralized entities, a requirement that has become increasingly concerning to society.

To combat this concern, a new Internet age has begun: web3. In the web3 version of the Internet, blockchains have emerged to provide decentralized, immutable ledgers that enable users to interact with one another securely and reliably, all without requiring trust in controlling intermediaries or centralized entities. Similar to how

web2 Internet services and applications rely on cloud infrastructure as building blocks, decentralized applications can use blockchains as a decentralized infrastructure layer to reach billions of users across the world.

However, despite the existence of many blockchains today, widespread adoption of web3 has not yet taken place. While technology continues to advance the industry, existing blockchains are unreliable, impose high transaction fees for users, have low throughput limitations, suffer regular asset losses due to security issues, and cannot support real-time responsiveness. In comparison to how cloud infrastructure has enabled web2 services to reach billions, blockchains have not yet enabled web3 applications to do the same.

1 Introduction

We present the RENEC blockchain, designed with scalability, safety, reliability, and upgradeability as key principles, to address these challenges. RENEC is a decentralized blockchain platform designed to enable creators to build experiences that provide to the billion users in the Web3 era. Founded in 2021, RENEC blockchain is an open-source project currently run by the RENEC foundation. RENEC's third-generation blockchain architecture is designed to facilitate smart contracts and decentralized application (DApp) creation. The RENEC genesis occurred on October 14, 2022. RENEC aims to scale throughput beyond what is typically achieved by popular blockchains while keeping costs low.

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 participants abilities to keep time. 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. RENEC implements an innovative hybrid consensus model that combines a unique proof-of-history (PoH) algorithm with the lightning-fast synchronization engine, which is a version of proof-of-stake (PoS).

Infrastructure concerns should fade into the background. Developers and users will have access to many different options for key recovery, data modeling, smart contract standards, resource usage tradeoffs, privacy, and composability. Users know that their assets are secure, always available, and can be accessed with near at-cost fees. Anyone can safely, easily, and immutably transact with untrusted parties worldwide. Blockchains are as ubiquitous as cloud infrastructure.

The RENEC vision is to deliver a blockchain that can bring mainstream adoption to web3 and empower an ecosystem of decentralized applications to solve real-world user problems. Our mission is to advance the state-of-the-art in blockchain reliability, safety, and performance by providing a flexible and modular blockchain architecture. This architecture should support frequent upgrades, fast adoption of the latest technology advancements, and first-class support for new and emerging use cases.

We envision a decentralized, secure, and scalable network governed and operated by the community that uses it. When infrastructure demands grow across the world, the computational resources of the blockchain scale up horizontally and vertically to meet those needs. As new use cases and technological advances arise, the network should frequently and seamlessly upgrade without interrupting users.

2 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 3 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 time-stamped 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 other's sequences. Horizontal scaling is discussed in depth in later sections.

2.1 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:

Table 1: 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:

Table 2: 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 2.1, 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

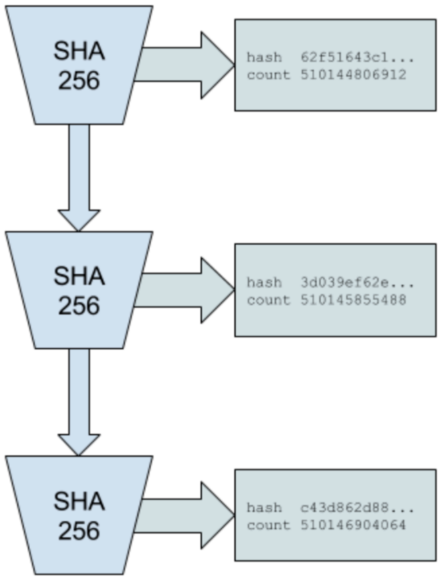


Figure 1: Proof of History Sequence

2.2 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:

Table 3: POH Sequence

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

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

Table 4: 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 3.3.

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.

Table 5: 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, `photograph1_sha256`))	hash336
400	sha256("hash399")	hash400
500	sha256("hash499")	hash500
600	sha256(append(hash599, `photograph2_sha256`))	hash600
700	sha256("hash699")	hash700

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 2.2, 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.

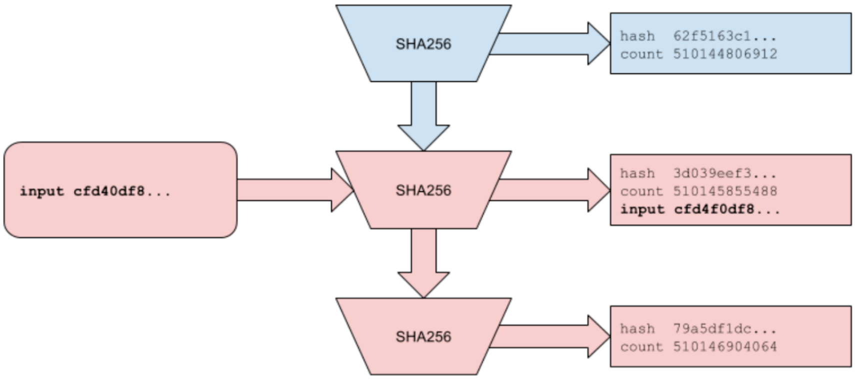


Figure 2: Insert data into PoH

2.3 Verification

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

For example:

Table 6: Core 1

Index	Operation	Output Hash
200	sha256("hash199")	hash200
300	sha256("hash299")	hash300

Table 7: Core 2

Index	Operation	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:

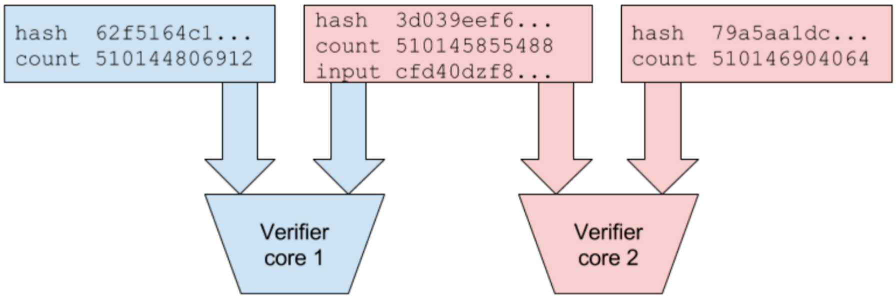


Figure 3: Multiple Core Verifications

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

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

$$\frac{\text{Total Number of hashes}}{\text{Hashes per second per core} * \text{Number of cores available for verify}}$$

In the example in Figure 2.3, 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.

2.4 Horizontal Scaling

It is 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.

Table 8: POH Generator A / B

Index	**Hash**	**Data**	**Index**	**Hash**	**Data**
1	hash1a		1	hash1b	
2	hash2a	hash1b	2	hash2b	hash1a
3	hash3a		3	hash3b	
4	hash4a		4	hash4b	

Table 9: PoH Senquence A / PoH *Hidden* Sequence B

Index	**Data**	**Output Hash**	**Index**	**Data**	**Output Hash**
10		hash10a	10		hash10b
20	Event1	hash20a	20	Event3	hash20b
30	Event2	hash30a	30	Event2	hash30b
40	Event3	hash40a	40	Event1	hash40b

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 $\longleftrightarrow$ B $\longleftrightarrow$ 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 2.5, 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 $\longleftrightarrow$ B $\longleftrightarrow$ 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 $0.99910 = 0.99$ availability.

2.5 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.

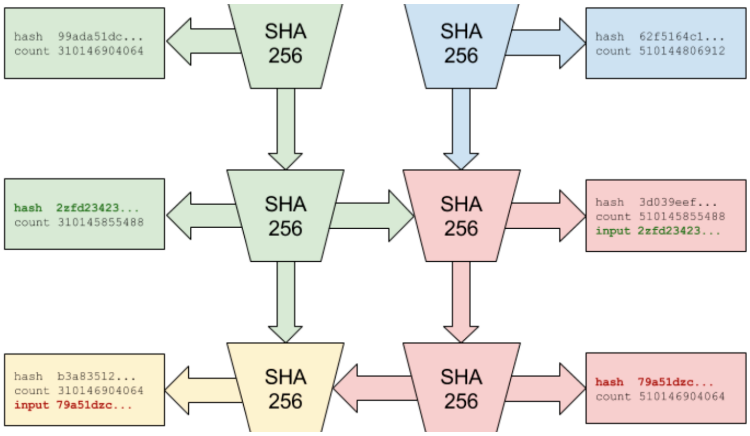


Figure 4: Two generators synchronizing

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.

Table 10: PoH Senquence 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.

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

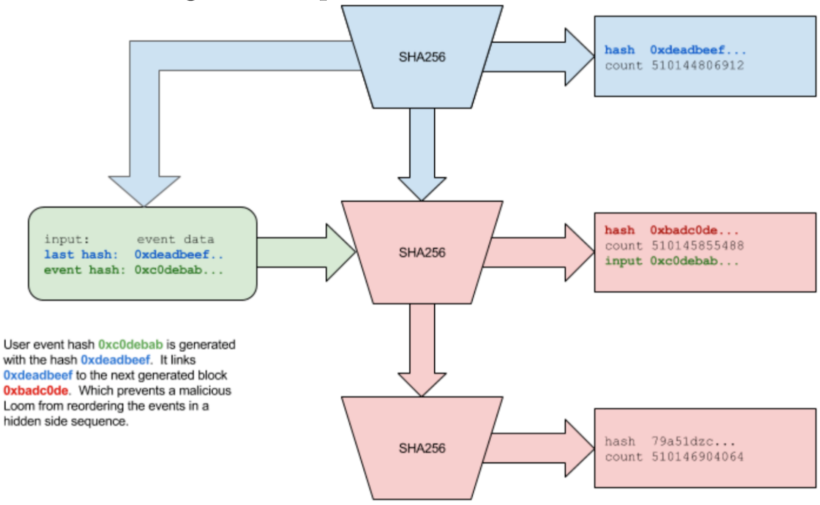
Table 11: PoH Senquence A

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

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

In Figure 2.5, the user-supplied input is dependent on hash 0xdeadbeef... existing in the generated sequence sometime before its inserted. The blue

Figure 5: 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.

2.6 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.

2.7 Attacks

2.7.1 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.

2.7.2 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.

2.7.3 Long Range Attacks

Long range attacks involve acquiring old discarded client Private Keys, and generating a falsified ledger. 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 the next version of this whitepaper). 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.

3 Proof of Stake Consensus

3.1 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.

3.2 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 a 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. 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.

supermajority A super majority is 2/3 rds of the validators weighted by their bonds. A super majority vote indicates that the network has reached consensus, and at least 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 1/3 rd of the market cap of the coin.

3.3 Bonding

A bonding transaction takes an 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 a super majority of the current stakeholders have confirmed the sequence.

3.4 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 a super majority of the bonded identities have voted within a timeout, then this branch would be accepted as valid.

3.5 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 the event of a large network partition, this allows the larger branch to recover faster than the smaller branch.

3.6 Elections

Election for a new PoH generator occurs 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 supermajority.

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 pre-defined schedule.

3.7 Election Triggers

3.7.1 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.

3.7.2 Runtime Exceptions

A hardware failure or a bug, or an 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.

3.7.3 Network Timeouts

A network timeout would trigger an election.

3.8 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.

3.9 Secondary Elections

Secondary and lower ranked Proof of History generators can be proposed and approved. A proposal is cast on the primary generator 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.

3.10 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 coins 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 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 2/3 rds but above 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 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 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.

3.11 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 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.

3.12 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.

3.13 Attacks

3.13.1 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.

3.13.2 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.

3.13.3 Censorship

Censorship or denial of service could occur when a 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. The 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.

3.13.4 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.

3.13.5 ASIC Attack

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 then 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 the 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 and having a single combined stake and only using 1 set of hardware.

4 System Architecture

4.1 Description

4.1.1 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 guarantees 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.

4.1.2 State

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.

Table 12: Transaction table

0 31 63 95 127 159	191 223	255
Ripemd of User Public Key	Account	Unused

for a total of 32 bytes

Table 13: Proof of Stake bonds table

0 31 63 95	127 159	191 223	255
Ripemd of User Public Key		Bond	
Last Vote			
Unused			

for a total of 64 bytes

4.1.3 Verifier, State Replication

The Verifier 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 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.

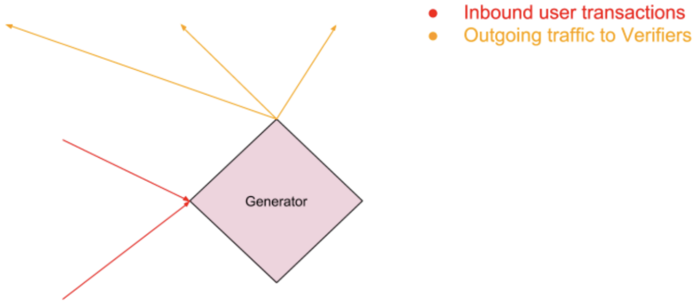
4.1.4 Validators

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.

4.1.5 Network Limits

Leader is expected to be able to take incoming user packets, orders them the most efficient way possible, and sequences them into a Proof of History sequence that is published to downstream Verifiers. Efficiency is based on

Figure 6: Generator Network Limits



memory access patterns of the transactions, so the transactions are ordered to minimize faults and to maximize prefetching.

Incoming packet format:

Table 14: Incoming packet format

0 31 63 95 127 159 191 223 255

Last valid Hash				Counter		u	s
Fee							
From							
Signature 1 of 2							
Signature 2 of 2							

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

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

Table 15: Payload packet format

0 31 63 95 127 159 191 223 255

Last valid Hash				Counter		u	s
To				Amount			
Counter		Fee					
From							
Signature 1 of 2							
Signature 2 of 2							

With minimum payload 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:

Table 16: Proof of History packet

0	31	63	95	127	159	191	223	255
Current Hash					Counter			
Message Hash								
State Hash								
Signature 1 of 2								
Signature 2 of 2								

On a 1gbps network connection the maximum number of transactions possible is 1 gigabit per second / 176 bytes = 710k tps max. Some loss of 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.

4.2 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.

4.3 Memory Limit

A naive implementation of the state as a 50 TODOpercent 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 TODO 107 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.

4.4 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 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

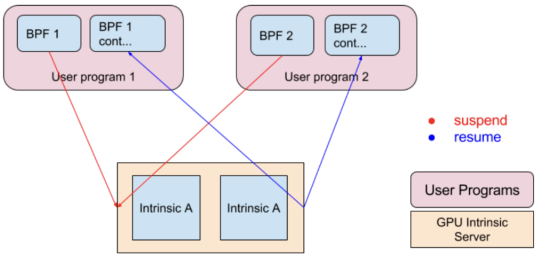


Figure 7: BPF Excecution

complete. An example intrinsic is ECDSA verification. Batching these calls to execute on the GPU can increase throughput by thousands of times.

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.

5 References

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 Oracle, 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

[11] Solana: A new architecture for a high performance blockchain v0.8.13, https://solana.com/solana-whitepaper.pdf

[12] The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure, https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf

6 Legal Disclaimer

Nothing in this White Paper is an offer to sell, or the solicitation of an offer to buy, any tokens. RENEC Foundations and its affiliates shall have no liability for damages of any kind arising from the use, reference to, or reliance on this white paper or any of the content contained herein, even if advised of the possibility of such damages. This paper is a description of the current and planned RENEC ecosystem, the participants designing and developing it, and the project undertaken to bring it to fruition.

As such, this paper may contain predictions, estimates, or other information that might be considered forward-looking. While these forward-looking statements represent RENEC Foundations's current assessment of what the future holds, they are subject to risks and uncertainties that could cause actual results to differ materially. Hence, the reader of this white paper is cautioned not to place undue reliance on these forward-looking statements, which reflect the opinions of the RENEC team only as of the date of issuance of the paper.

Please bear in mind that RENEC Foundation does not obligate itself to revise or publicly release the results of any revisions to these forward-looking statements in light of new information or future events.