Endgame Architecture

Solana's goal is to synchronize a single permissionless global state machine as fast as the laws of physics allow. I believe the architecture that can achieve that goal will look like this:

- Many full nodes, more than 10,000 (N > 10,000)

For the network to function as a global state machine, it needs to support numerous full nodes. Turbine has already demonstrated that rapid replication to a very large network is scalable on modern hardware and networks.

- Many block-producing leaders, more than 10,000 (N > 10,000)
- Concurrent leaders simultaneously producing blocks, randomly selected, in the range of 4 to 16.

Concurrent leaders enable the network to have multiple locations worldwide capable of sequencing user transactions. This reduces the distance between the user and the network, eliminating the need for a full global round trip before the transaction is added to the chain.

- Block times of 120ms

Short block times create rapid points of finality, enhance censorship resistance, improve user experience, reduce the window for reordering transactions, and overall accelerate the network.

- Some voting consensus nodes in the quorum subcommittee, numbering between 200 and 400, chosen randomly from leaders and rotated every epoch, which should be 4 to 8 hours.

Consensus is essential for selecting forks, and forks occur due to network partitions. A sample of 200 or more nodes will statistically represent all the major partitions in the network, closely matching their actual distributions. Therefore, it is unnecessary for all full nodes to vote, and 200 is more than sufficient. Limiting the quorum to a subcommittee reduces the memory and network bandwidth required to support 120ms blocks. Reducing block times naturally increases the number of votes sent per second, straining the resources allocated for consensus.

The real challenge with 120ms blocks is replaying all user transactions. Because the network is permissionless, guaranteeing a homogenous execution environment with reliable timing for executing arbitrary user code is extremely difficult. It is possible, but only by limiting the available computing resources for user transactions and ensuring that every node is over-provisioned for the worst-case scenario.

However, there is no reason for consensus nodes that vote on forks or leaders building on forks to execute the full state. To keep the quorum of consensus nodes and leaders synchronized, the state only needs to be computed once an epoch.

Asynchronous Execution

Motivation

Synchronous execution requires all the nodes that vote and create blocks to be over provisioned for the worst case execution time in any block. Asynchronous execution is one of the rare cases where there are virtually no trade-offs. Consensus nodes can perform less work before voting. Work can be aggregated and batched, making it efficiently executed without any cache misses. It can even be executed on a different machine altogether from the consensus nodes or leaders. Users who desire synchronous execution can allocate enough hardware resources to execute every state transition in real time without waiting for the rest of the network.

Given the diversity of applications and core developers, it is worth planning a single major protocol change per year. If we have to choose one, my vote would be for Asynchronous Execution.

Overview

Currently, validators replay all transactions as fast as possible on every block and only vote once the full state is computed for the block. The goal of this proposal is to separate the decision to vote on a fork from computing the full state transition for the block.

Validators voting in the quorum only need to select forks; they do not need to execute any state at all. It is only once per epoch when they need the state to compute the next quorum.

The vote program is adjusted so that it can be executed independently. Nodes only execute the vote program before voting. Memory requirements should be relatively small since validators do not occupy much space. Since votes have a very predictable execution time, vote program execution should complete virtually without any jitter.

All the non-vote transactions can be computed asynchronously. This allows replay to batch execute all the non-vote transactions, prefetch, and JIT all the programs ahead of time, virtually eliminating all cache misses. The long-term goal is that only the machines needing real-time low-latency full state computation are provisioned for this task. Presumably, users will pay for the extra hardware.

Once fork choice and state execution are separated, it is much easier to make everything faster:

1. Asynchronous execution
2. Fixed-size voting subcommittees rotated every epoch
3. 200ms block times

Since user transaction replay cannot block fork choice, replay jitter is no longer a concern when reducing block times. The only thing to consider is that the validator vote rate is doubled at 200ms. A fairly straightforward change to how the quorum is computed would allow us to fix the size of the quorum to 200 or 400 or whatever number seems appropriate.

It is also natural to separate execution from consensus as separate instances altogether. Restarting a consensus node that only needs to check the vote program accounts in a fixed-size quorum is going to be much faster.

In practice, I believe that confirmation times will improve because the quorum supermajority will vote as fast as possible, and while those votes propagate, the node providing full state execution results to the user can execute transactions concurrently. So any replay jitter that we see today should overlap with the network propagation of votes.

Voting

- Vote accounts must have enough SOL in them to cover 2 epochs worth of votes.
- Vote transactions must be simple votes. Non simple votes must fail execution. Block producers should drop complex votes.
- Withdrawing SOL from the vote account is allowed as long as the balance doesn't go below 1 epochs worth of votes.
- To remove all the lamports, Vote CLOSE instruction must require a full epoch to pass. Vote accounts are marked for CLOSE on epoch 1, but can only be CLOSE on epoch 2. CLOSE allows all the sol to be withdrawn and for the vote account to be deleted. Once an account is marked for CLOSE it can only be fully deleted, it cannot be reopened.
- Votes contain a VoteBankHash instead of regular BankHash

Leader Scheduler and Quorum

Only validators with:

- stake > X
- AND SOL > 2 epochs worth of votes
- AND not marked to CLOSE

are in the leader scheduler and count towards the quorum. For V2, we can separate the LeaderSchedule from the Quorum, the requirements to be in each one do not have to be the same.

VoteBankHash computation

Instead of computing a Bankhash for all transactions, validators only calculate a VoteBankHash for simple vote transactions pertaining to validators in the LeaderScheduler. All other transactions are disregarded. After replaying all the votes, the VoteBankHash is computed in the same format as the current BankHash.

VoteBankHash should accumulate the previous VoteBankHash and not the full BankHash.

BankHash computation

For all optimistically confirmed blocks (configurable to all blocks), validators begin computing the UserBankHash, which includes all state transitions except for the transactions already considered in the VoteBankHash computation.

The BankHash is then derived from the accumulation of (VoteBankHash, UserBankHash). The top 99.5% of validators submit the BankHash as part of their vote every 100th slot. While it is submitted every 100th slot, it is computed on every slot. It's probably worth it for some small percentage of nodes to always submit the BankHash in gossip as a soft signal that no non-determinism has been observed.

If fewer than 67% of validators submit the full BankHash computation, leaders should reduce the available block space for user transactions and writable accounts by 50%. This measure is in place to safeguard the chain from exploits that could excessively increase replay time.

BankHash should accumulate the previous BankHash as well.

Bankless Leader

During block creation, it's unlikely that the state will be available for the leader to create blocks, and it's not desirable to execute all the transactions during block creation.

- Leader maintains a cache of fee paying account balances
- If a fee payer is used as a writable account in as a system transfer source or passed as a writable account along with the system program to another program the fee paying balance is set to 0
- Pack the blocks based on declared CUs until the block is full, based on local fee priority ordering
- Subtract the fee from the fee payer balance cache
- Fee paying account balance cache is replenished by the BankHash computation

The cost to the network of failed transaction spam is relatively small, comprising only the bytes stored in the archive and the bandwidth required to propagate the transactions in the block.

Considering that validators already seek to maximize their own earnings, there is ample incentive for them to maintain an accurate cache of fee-paying accounts. Furthermore, if there is no slashing mechanism in place, it would become straightforward over the long term for the cache to be served by anyone in the network. In the event of server corruption, the bankless leader operator should be able to switch easily or sample multiple sources.

Tradeoffs

The main trade-off is that full nodes serving the user state lack a signature confirming that the state they provide matches exactly what the rest of the quorum computed. There exists only one canonical interpretation of the state, which should remain unchanged even if each transaction is replayed sequentially in the ledger. Any performance optimizations should not alter the outcome. Thus, once forks are finalized, there remains only one correct state that can be computed, as long as the runtime implementation remains free of bugs.

Nodes aiming to provide the state reliably should operate multiple machines and clients, and they should halt operations if discrepancies in state execution arise. This is essentially what operators should be doing today since relying solely on the rest of the network introduces honest majority assumptions.

Users can also sign transactions that assert the BankHash or trigger an abort. These transactions will only be executed by the rest of the network if they calculate the exact same BankHash as the one provided to the user by the RPC provider.

Long Term Roadmap to Stateless Consensus nodes

A network with a fixed sized quorum only needs a very small amount of state to start. The quorum itself with the stake weights, and all the vote account balances. This is a very small amount of memory, and a tiny snapshot file that can be rapidly distributed and quickly initialized on restart.

Full nodes that are following both the quorum and the state will halt if the quorum ever disagrees with the full nodes. That means, exchanges, fiat ramps, RPCs, bridges, etc.. will halt if the quorum ever diverges from the state. It would take a super-minority of faulty stateless consensus nodes for that to occur.

Bankless leaders can rely on a sample of multiple full nodes to provide the initial cache of account balances for fee payers. Even if faulty, the result would be spam in blocks and not a consensus failure. Operators should be able to monitor the health of their leaders and what percentage spam they are injecting into blocks and quickly respond to failures.