HyperLedger Sawtooth Lake

HyperLedger

Umbrella project of open source blockchains by the Linux Foundation, to support the collaborative development of blockchain-based distributed ledgers.

Examples

- HyperLedger Burrow
- HyperLedger Sawtooth
- HyperLedger Fabric
- HyperLedger Indy
- HyperLedger Iroha

HyperLedger Sawtooth

Hyperledger Sawtooth is an enterprise blockchain platform for building distributed ledger applications and networks. The design philosophy targets keeping ledgers distributed and making smart contracts safe, particularly for enterprise use.

Distinctive Features

- Separation Between the Application Level and the Core System
- Transaction Families
- Pluggable consensus algorithms
- Private Networks with the Sawtooth Permissioning Features
- Parallel Transaction Execution

Separation Between the Application Level and the Core System

- Application in any language, separate from core system
- Application can be in native business language or on Smart Contract Virtual Machine
- Separate transaction processing layer, hence multiple types of transaction on the same chain

Transaction Family

A Transaction Family is a group of operations or transaction types that you allow on your ledger. Transaction families differ by

- Type of transactions
- Number of operations
- Risks allowed

Examples of Transaction Families

- XO Transaction Family
- MarketPlace Transaction Family
- Bond Transaction Family
- Integer Key Transaction Family

Pluggable Consensus Algorithms

- Sawtooth abstracts the core concepts of consensus and isolates consensus from transaction semantics
- Sawtooth allows different types of consensus on the same blockchain
- Sawtooth currently supports
  - Proof Of Elapsed Time (PoET)
  - PoET simulator
  - Dev mode - Random election algorithm

Private Networks with the Sawtooth Permissioning Features

- Clusters of Sawtooth nodes can be easily deployed with separate permissioning
- The blockchain stores the settings that specify the permissions, such as roles and identities, so that all participants in the network can access this information

Parallel Transaction Execution

- Splits transactions into parallel flows
- Isolates the execution of transactions from one another while maintaining contextual changes
- Transactions are executed in parallel, while preventing double-spending even with multiple modifications to the same state

Proof of Elapsed Time (POET)

- POET is a consensus mechanism used to decide mining rights or block winners on the network.

- It utilizes a trusted execution environment to improve on the efficiency of present solutions such as proof of work.

Working

- Each validator requests a wait time from a trusted function to determine how long it has to wait before it is allowed to generate a block.

- One function, such as “CreateTimer”, creates a timer for a transaction block.

- Validator with the shortest wait time is elected the leader.

Continued

- Another function, such as “CheckTimer”, verifies that the timer was created by the trusted function.

- POET relies on special CPU Instruction set called Intel’s SGX. Intel SGX is a set of central processing unit (CPU) instruction codes from Intel.

Two Conditions that are to be met by Intel SGX

- Specialized Hardware can create an attestation that allows a Participant to prove to other participants that it is running the right trusted code in a TEE.

- Trusted code runs in an environment private to the application that is a TEE. It ensures that a malicious program can’t cheat by manipulating POET’s trusted code.

Lottery function properties and how it is achieved

- Fairness
- Investment
- Verification
- POET achieves these goals using Intel’ s SGX

Terms used -

- Each wait certificate will have -
  - Duration - Total amount of time the validator had to wait before generating the corresponding block
  - Local avg wait - Estimated local average of the wait time at that time
- Min_wait - Lower bound on the waiting time
- b - no of blocks currently stored on the ledger
- Target wait time - Desired average wait time which depends on the Network Diameter
- Sample Length - Min no of blocks required which is a system parameter

Calculation of local avg wait time -

For each wait certificate {

sw=sw+duration - min_wait ; sm=sm+local avg wait ; }

ps=sm/sw;

If (b < sample length) {

r=(1.0*b)/sample length;

Local avg wait = target wait time *(1-r*r) + initial wait time * (r*r); }

Else {

Local avg wait = target wait time * ps;

}

Calculation of wait time

Wait time = Min_wait - local avg wait * log(r)

r is in the range [0,1] and is derived from the hash of the node's previous wait certificate

Purpose of local avg wait is to adjust wait time according to the no of active nodes.

Z- test

A z-test is used to test the hypothesis that a validator won elections at a higher average rate than expected.

Assume-

1. Each node has same winning probability p
2. m is the total no of blocks generated so far

Win_num is the no of blocks the node has successfully generated

Continued

X is the no of winning times

X ~ N(mp,sqrt(mp(1-p))

Z_score = (win_num - mp)/sqrt(mp(1-p))

If Z_score > Zmax(a system parameter) the hypothesis is satisifed

Using Z test we can detect a potentially compromised node that produces blocks at a higher rate.

Architecture Description

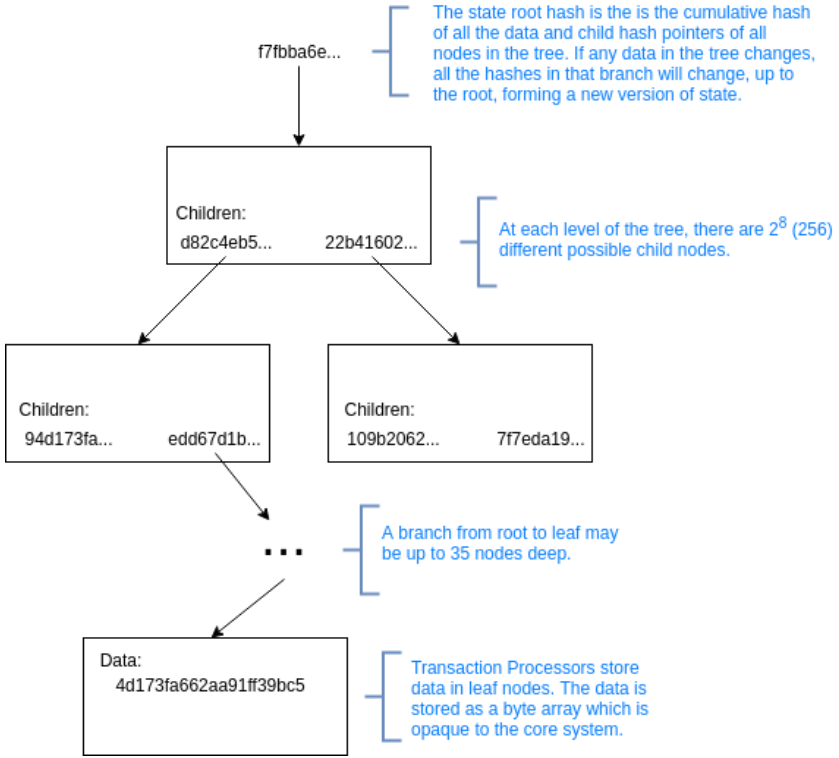
- Global State
- Transactions and Batches
- Journal
- Transaction Scheduling

Global State

Radix Merkle Tree

States of all transaction families are stored in a radix merkle tree.

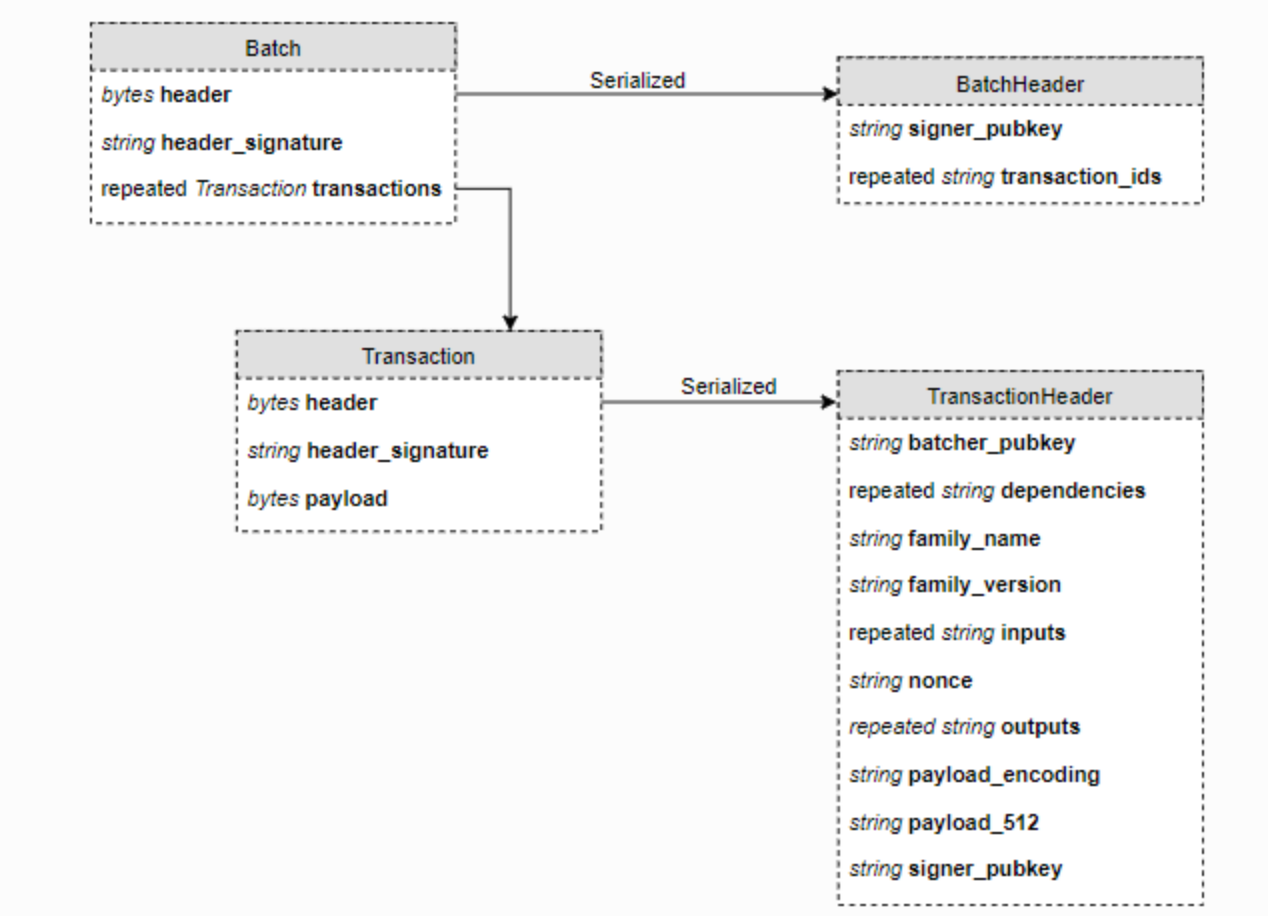
A copy of this tree is with each validator.



Transactions and Batches

Modifications to state are performed by creating and applying transactions. A client creates a transaction and submits it to the validator. The validator applies the transaction which causes a change to state.

Transactions are always wrapped inside of a batch. All transactions within a batch are committed to state together or not at all. Thus, batches are the atomic unit of state change.

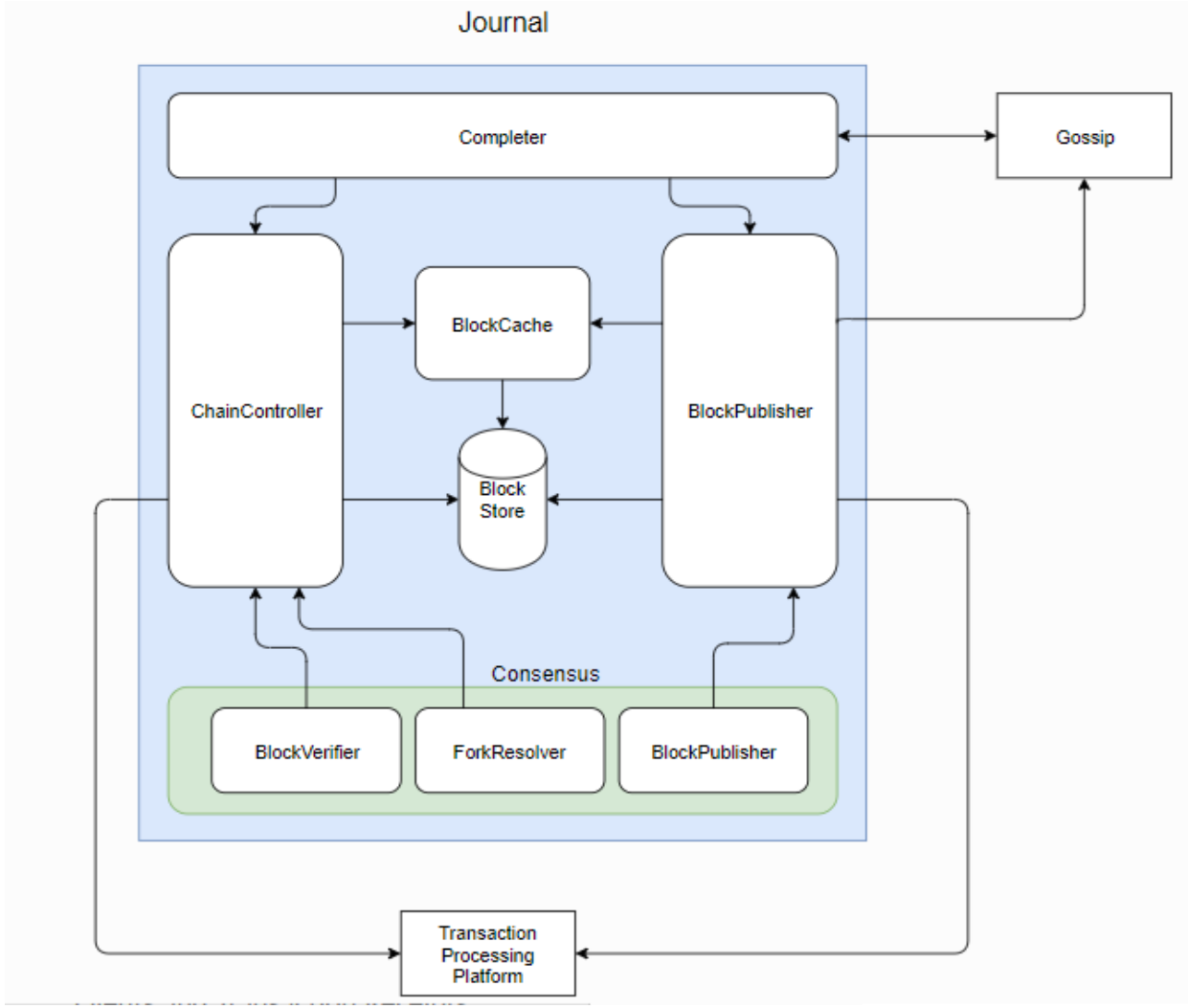

Batch
bytes header
string header_signature
repeated Transaction transactions
Serialized
BatchHeader
string signer_pubkey
repeated string transaction_ids
Transaction
bytes header
string header_signature
bytes payload
Serialized
TransactionHeader
string batcher_pubkey
repeated string dependencies
string family_name
string family_version
repeated string inputs
string nonce
repeated string outputs
string payload_encoding
string payload_512
string signer_pubkey

Prevent Double Spending

There is an important restriction enforced between transactions and batches, which is that the transaction must contain the public key of the batch signer in the batcher_pubkey field. This is to prevent transactions from being reused separate from the intended batch. So, for example, unless you have the batcher’s private key, it is not possible to take transactions from a batch and repackage them into a new batch, omitting some of the transactions.

JOURNAL

- Responsible for maintaining and extending blockchain
- Consumer of the blocks and batches that arrive at the validator
- Responsibilities -
  - Validating candidate blocks
  - Evaluating valid blocks
  - Generating new blocks to extend the chain


Journal
Completer
Gossip
ChainController
BlockCache
BlockPublisher
Block
Store
Consensus
BlockVerifier
ForkResolver
BlockPublisher
Transaction
Processing
Platform

The BlockStore

- Contains all the blocks in the current blockchain
- Blocks from forks not included in it.
- Provides atomic means to update the store in case of a current fork change.
- maintains internal mappings of Transaction-to-Block and Batch-to-Block
- An error in block store is considered non recoverable error for the validator.

The BlockCache

- Holds working set of the blocks for the validator
- Tracks processing state as valid, invalid, or unknown

The Completer

- Responsible for making sure Blocks and Batches are complete before they are delivered.
- All Blocks and Batches will have a timeout for being completed.

The Consensus Interface

Journal supports pluggable consensus algorithms that may be changed via setting transactions family. Divided into three interfaces:-

1. Consensus.BlockPublisher
2. Consensus.BlockVerifier
3. Consensus.ForkResolver

The ChainController

- Responsible for determining which chain the validator is currently on and coordinating any change-of-chain activities that need to happen.
- Designed to be able to handle multiple block validation activities simultaneously.
- When the chain needs to be updated, the ChainController does an update of the ChainHead using the BlockStore
- Serialize block validation when any block is received and any of its predecessors are still being validated

The BlockValidator

Subcomponent of chain controller responsible for Block validation and fork resolution. It has following stages of evaluation :-

- Determine the common root of the fork (ForkRoot).
- The Candidate chain is validated. This process walks forward from the ForkRoot and applies block validation rules (described below) to each Block successively.
- Fork resolution requires a determination to be made if the Candidate should replace the ChainHead and is deferred entirely to the consensus implementation. If the block is the new ChainHead, the answer is returned to the ChainController, which updates the BlockStore.

Block Validation

Block validation has the following steps that are always run in order. Failure of any validation step results in failure, processing is stopped, and the Block is marked as Invalid:-

1. **Transaction Permissioning**
2. **On-chain Block Validation Rules**
3. **Batches Validation**
4. **Consensus Verification**
5. **State Hash Check**

The BlockPublisher

- responsible for creating candidate blocks to extend the current chain.
- takes direction from the consensus algorithm for when to create a block and when to publish a block.