Sawtooth Documentation

Release latest

Intel Corporation

Mar 13, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

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.

Sawtooth simplifies blockchain application development by separating the core system from the application domain. Application developers can specify the business rules appropriate for their application, using the language of their choice, without needing to know the underlying design of the core system.

Sawtooth is also highly modular. This modularity enables enterprises and consortia to make policy decisions that they are best equipped to make. Sawtooth's core design allows applications to choose the transaction rules, permissioning, and consensus algorithms that support their unique business needs.

Sawtooth is an open source project under the Hyperledger umbrella. For information on how to contribute, see *Join the Sawtooth Community*.

1.1 About Distributed Ledgers

A "distributed ledger" is another term for a blockchain. It distributes a database (a ledger) of transactions to all participants in a network (also called "peers" or "nodes"). There is no central administrator or centralised data storage. In essence, it is:

- **Distributed**: The blockchain database is shared among potentially untrusted participants and is demonstrably identical on all nodes in the network. All participants have the same information.
- **Immutable**: The blockchain database is an unalterable history of all transactions that uses block hashes to make it easy to detect and prevent attempts to alter the history.
- **Secure**: All changes are performed by transactions that are signed by known identities.

These features work together, along with agreed-upon consensus mechanisms, to provide "adversarial trust" among all participants in a blockchain network.

1.2 Distinctive Features of Sawtooth

1.2.1 Separation Between the Application Level and the Core System

Sawtooth makes it easy to develop and deploy an application by providing a clear separation between the application level and the core system level. Sawtooth provides smart contract abstraction that allows application developers to write contract logic in a language of their choice.

An application can be a native business logic or a smart contract virtual machine. In fact, both types of applications can co-exist on the same blockchain. Sawtooth allows these design decisions to be made in the transaction-processing layer, which allows multiple types of applications to exist in the same instance of the blockchain network.

Each application defines the custom *transaction processors* for its unique requirements. Sawtooth provides several example *transaction families* to serve as models for low-level functions (such as maintaining chain-wide settings and storing on-chain permissions) and for specific applications such as performance analysis and storing block information.

Transaction processor SDKs are available in multiple languages to streamline creation of new contract languages, including Python, JavaScript, Go, C++, Java, and Rust. A provided REST API simplifies client development by adapting *validator* communication to standard HTTP/JSON.

1.2.2 Private Networks with the Sawtooth Permissioning Features

Sawtooth is built to solve the challenges of permissioned (private) networks. Clusters of Sawtooth nodes can be easily deployed with separate permissioning. There is no centralized service that could potentially leak transaction patterns or other confidential information.

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.

1.2.3 Parallel Transaction Execution

Most blockchains require serial transaction execution in order to guarantee consistent ordering at each node on the network. Sawtooth includes an advanced parallel scheduler that splits transactions into parallel flows. Based on the locations in state which are accessed by a transaction, Sawtooth isolates the execution of transactions from one another while maintaining contextual changes.

When possible, transactions are executed in parallel, while preventing double-spending even with multiple modifications to the same state. Parallel scheduling provides a substantial potential increase in performance over serial execution.

1.2.4 Event System

Hyperledger Sawtooth supports creating and broadcasting events. This allows applications to:

- Subscribe to events that occur related to the blockchain, such as a new block being committed or switching to a new fork.
- Subscribe to application specific events defined by a transaction family.
- Relay information about the execution of a transaction back to clients without storing that data in state.

Subscriptions are submitted and serviced over a ZMQ Socket.

1.2.5 Ethereum Contract Compatibility with Seth

The Sawtooth-Ethereum integration project, Seth, extends the interoperability of the Sawtooth platform to Ethereum. EVM (Ethereum Virtual Machine) smart contracts can be deployed to Sawtooth using the Seth transaction family.

1.2.6 Dynamic Consensus Algorithms

In a blockchain, consensus is the process of building agreement among a group of mutually distrusting participants. Algorithms for achieving consensus with arbitrary faults generally require some form of voting among a known set of participants. General approaches include Nakamoto-style consensus, which elects a leader through some form of lottery, and variants of the traditional Byzantine Fault Tolerance (BFT) algorithms, which use multiple rounds of explicit votes to achieve consensus.

Sawtooth abstracts the core concepts of consensus and isolates consensus from transaction semantics. The interface supports plugging in various consensus implementations. More importantly, Sawtooth allows different types of consensus on the same blockchain. The consensus is selected during the initial network setup and can be changed on a running blockchain with a transaction.

Sawtooth currently supports these consensus implementations:

- Proof of Elapsed Time (PoET), a Nakamoto-style consensus algorithm that is designed to be a production-grade protocol capable of supporting large network populations. PoET relies on secure instruction execution to achieve the scaling benefits of a Nakamoto-style consensus algorithm without the power consumption drawbacks of the Proof of Work algorithm.
- PoET simulator, which provides PoET-style consensus on any type of hardware, including a virtualized cloud environment.
- Dev mode, a simplified random-leader algorithm that is useful for development and testing.

1.2.7 Sample Transaction Families

In Sawtooth, the data model and transaction language are implemented in a *transaction family*. While we expect users to build custom transaction families that reflect the unique requirements of their ledgers, we provide several core transaction families as models:

- IntegerKey - Used for testing deployed ledgers.
- Settings - Provides a reference implementation for storing *on-chain configuration settings*.
- Identity - Handles on-chain permissioning for transactor and validator keys to streamline managing identities for lists of public keys.

Additional transaction families provide models for specific areas:

- Smallbank - Handles performance analysis for benchmarking and performance testing when comparing the performance of blockchain systems. This transaction family is based on the H-Store Smallbank benchmark.
- BlockInfo - Provides a methodology for storing information about a configurable number of historic blocks.

For more information, see *Transaction Family Specifications*.

1.3 Real-world Application Examples

- XO: Demonstrates how to construct basic transactions by playing Tic-tac-toe. The XO transaction family includes create and take transactions, with an `xo` command that allows two participants to play the game. For more information, see *Introduction to the XO Transaction Family*.
- Sawtooth Supply Chain: Demonstrates how to trace the provenance and other contextual information of any asset. Supply Chain provides an example application with a transaction processor, custom REST API, and web app. This example application also demonstrates a decentralized solution for in-browser transaction signing, and illustrates how to synchronize the blockchain state to a local database for complex queries. For more information, see the sawtooth-supply-chain repository on GitHub.

- Sawtooth Marketplace: Demonstrates how to exchange specific quantities of customized assets with other users on the blockchain. This example application contains a number of components that, together with a Sawtooth validator, will run a Sawtooth blockchain and provide a simple RESTful API to interact with it. For more information, see the sawtooth-marketplace repository on GitHub.
- Sawtooth Private UTXO: Demonstrates how assets can be created and traded. This example application shows how to use SGX to allow for assets to be transferred off ledger and privately traded, where only the trading parties know the details of the transaction. For more information, see the sawtooth-private-utxo repository on GitHub.

1.4 Getting Started with Application Development

1.4.1 Try Hyperledger Sawtooth

The Sawtooth documentation explains how to set up a local *validator* for demonstrating Sawtooth functionality and testing an application. Once running, you will be able to submit new transactions and fetch the resulting state and block data from the blockchain using HTTP and the Sawtooth *REST API*. These methods apply to the included example *transaction families*, as well as to any transaction families you might write yourself.

Sawtooth validators can be run from pre-built Docker containers, installed natively using Ubuntu 16.04, or launched in AWS from the AWS Marketplace.

To get started, see *Installing and Running Sawtooth*.

1.4.2 Develop a Custom Application

In Sawtooth, the data model and transaction language are implemented in a transaction family. Transaction families codify business rules used to modify state, while client programs typically submit transactions and view state. You can build custom transaction families that reflect your unique requirements, using the provided core transaction families as models.

Sawtooth provides a REST API and SDKs in several languages - including Python, C++, Go, Java, JavaScript, and Rust - for development of applications which run on top of the Sawtooth platform. In addition, you can write smart contracts in Solidity for use with the Seth transaction family.

For more information, see *Application Developer's Guide*, *SDK API Reference*, and *REST API Reference*.

1.5 Participating in Core Development

1.5.1 Learn about Sawtooth Architecture

See *Architecture Description* for information on *Sawtooth core* features such as *global state*, transactions and *batches* (the atomic unit of state change in Sawtooth), permissioning, the validator network, the event system, and more.

1.5.2 Get the Sawtooth Software

The Sawtooth software is distributed as source code with an Apache license. You can get the code to start building your own distributed ledger.

- sawtooth-core: Contains fundamental classes used throughout the Sawtooth project, as well as the following items:

  - The implementation of the validator process which runs on each node
  - SDKs for writing transaction processing or validation logic in a variety of languages
  - Dockerfiles to support development or launching a network of validators
  - Source files for this documentation
- Seth: Deploy Ethereum Virtual Machine (EVM) smart contracts to Sawtooth
- Sawtooth Marketplace: Exchange customized "Assets" with other users on the blockchain
- Sawtooth Supply Chain: Trace the provenance and other contextual information of any asset
- Sawtooth Private UTXO: Create and trade assets, using SGX to allow assets to be transferred off-ledger and traded privately

1.5.3 Join the Sawtooth Community

Sawtooth is an open source project under the Hyperledger umbrella. We welcome working with individuals and companies interested in advancing distributed ledger technology. Please see *Community* for ways to become a part of the Sawtooth community.

1.6 Acknowledgements

This project uses software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

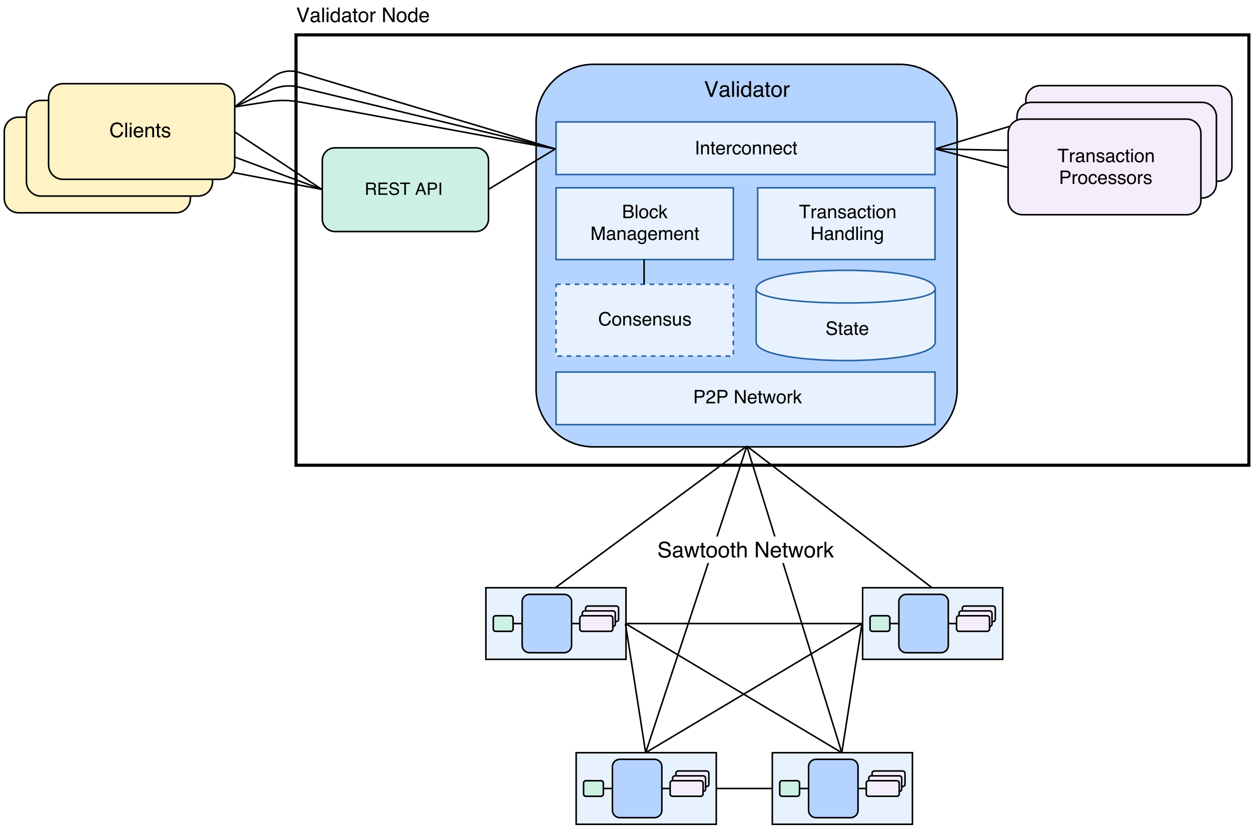
This project relies on other third-party components. For details, see the LICENSE and NOTICES files in the sawtooth-core repository.

CHAPTER

TWO

ARCHITECTURE DESCRIPTION

The following diagram shows a high-level view of the Sawtooth architecture.



2.1 Global State

One goal of a distributed ledger like Sawtooth, indeed the *defining* goal, is to distribute a ledger among participating nodes. The ability to ensure a consistent copy of data amongst nodes in Byzantine consensus is one of the core strengths of blockchain technology.

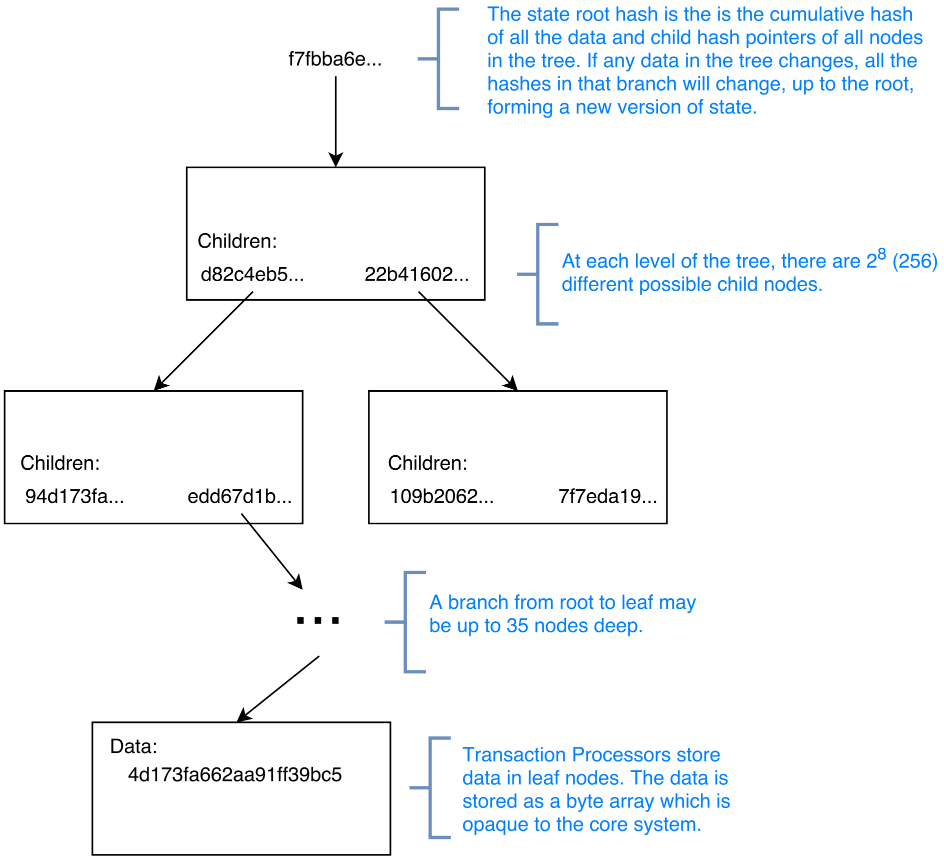
Sawtooth represents state for all transaction families in a single instance of a Merkle-Radix tree on each validator. The process of block validation on each validator ensures that the same transactions result in the same state transitions and that the resulting data is the same for all participants in the network.

The state is split into namespaces which allow flexibility for transaction family authors to define, share, and reuse global state data between transaction processors.

2.1.1 Merkle-Radix Tree Overview

Merkle Hashes

Sawtooth uses an addressable Merkle-Radix tree to store data for transaction families. Let's break that down: The tree is a Merkle tree because it is a copy-on-write data structure which stores successive node hashes from leaf-to-root upon any changes to the tree. For a given set of state transitions associated with a block, we can generate a single root hash which points to that *version* of the tree. By placing this state root hash on the block header, we can gain consensus on the expected version of state *in addition to* the consensus on the chain of blocks. If a validator's state transitions for a block result in a different hash, the block is not considered valid. For more information about general concepts, see the Merkle page on Wikipedia.



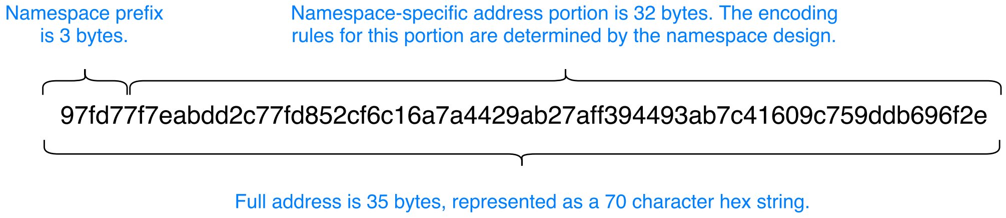
The state root hash is the is the cumulative hash of all the data and child hash pointers of all nodes in the tree. If any data in the tree changes, all the hashes in that branch will change, up to the root, forming a new version of state.

At each level of the tree, there are 2^8 (256) different possible child nodes.

A branch from root to leaf may be up to 35 nodes deep.

Transaction Processors store data in leaf nodes. The data is stored as a byte array which is opaque to the core system.

Radix Addresses

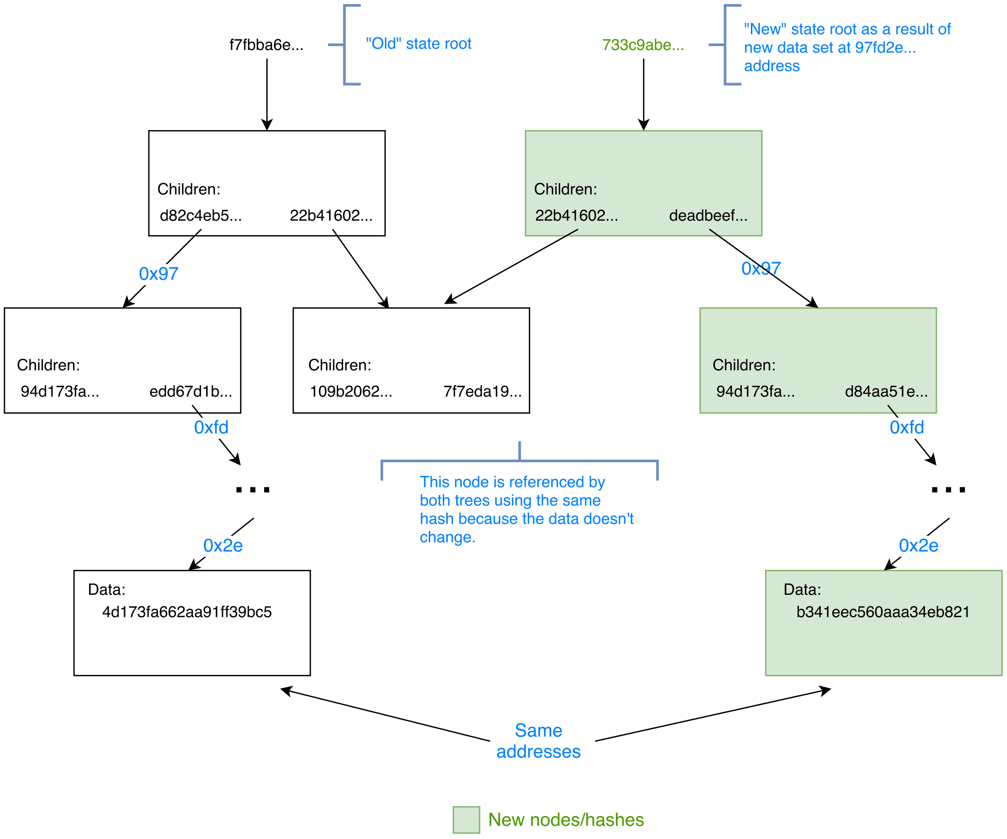

97fd77f7eabdd2c77fd852cf6c16a7a4429ab27aff394493ab7c41609c759ddb696f2e

Namespace prefix is 3 bytes.

Namespace-specific address portion is 32 bytes. The encoding rules for this portion are determined by the namespace design.

Full address is 35 bytes, represented as a 70 character hex string.

The tree is an addressable Radix tree because addresses uniquely identify the paths to leaf nodes in the tree where information is stored. An address is a hex-encoded 70 character string representing 35 bytes. In the tree implementation, each byte is a Radix path segment which identifies the next node in the path to the leaf containing the data associated with the address. The address format contains a 3 byte (6 hex character) namespace prefix which provides 2^{24} (16,777,216) possible different namespaces in a given instance of Sawtooth. The remaining 32 bytes (64 hex characters) are encoded based on the specifications of the designer of the namespace, and may include schemes for subdividing further, distinguishing object types, and mapping domain-specific unique identifiers into portions of the address. For more information about general concepts, see the Radix page on Wikipedia.



2.1.2 Serialization Concerns

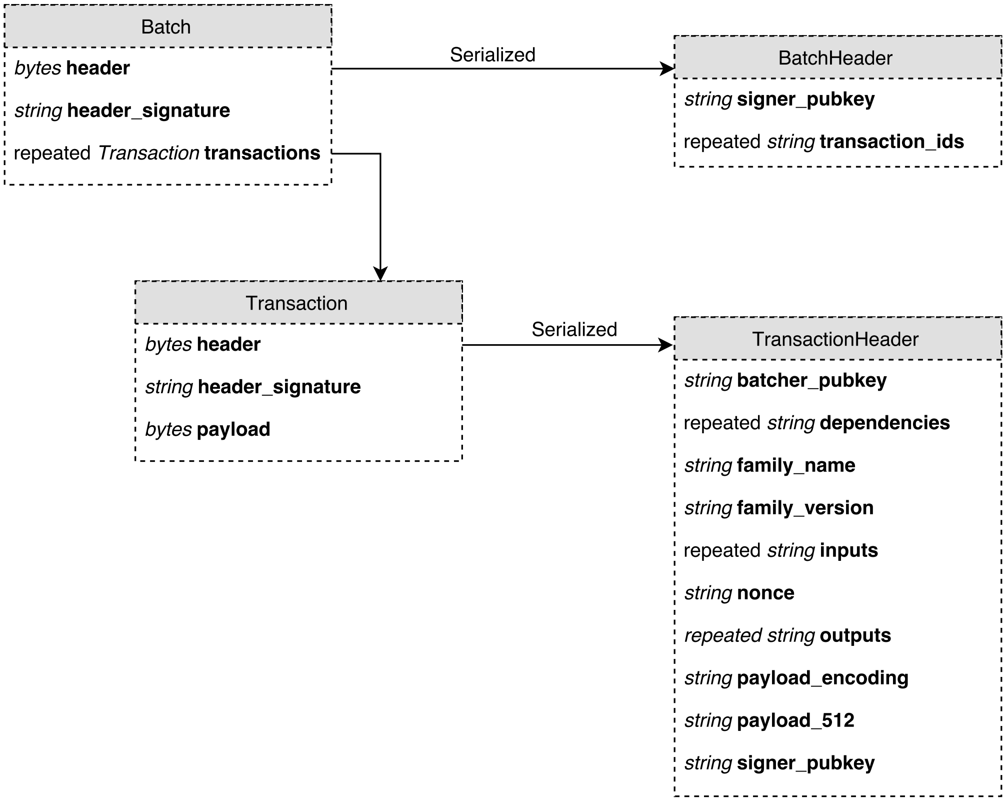
In addition to questions regarding the encoding of addresses, namespace designers also need to define the mechanism of serialization and the rules for serializing/deserializing the data stored at addresses. The domain-specific Transaction Processor makes get(address) and set(address, data) calls against a version of state that the validator provides. get(address) returns the byte array found at that address and set(address, data) sets the byte array stored at that address. The byte array is opaque to the core system. It only has meaning when deserialized by a domain-specific component based on the rules of the namespace. It is critical to select a serialization scheme which is deterministic across executions of the transaction, across platforms, and across versions of the serialization framework. Data structures which don't enforce ordered serialization (e.g. sets, maps, dicts) should be avoided. The requirement is to consistently produce the same byte array across space and time. If the same byte array is not produced, the leaf node hash containing the data will differ, as will every parent node back to the root. This will result in transactions and the blocks that contain them being considered valid on some validators and invalid on others, depending on the non-deterministic behavior. This is considered bad form.

2.2 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.

The overall structure of batches and transactions includes Batch, BatchHeader, Transaction, and TransactionHeader:


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**

2.2.1 Transaction Data Structure

Transactions are serialized using Protocol Buffers. They consists of two message types:

Listing 1: File: protos/transaction.proto

```
// Copyright 2016 Intel Corporation
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
```

(continues on next page)

(continued from previous page)

```
// limitations under the License.
// -----------------------------------------------------------------------------

syntax = "proto3";

option java_multiple_files = true;
option java_package = "sawtooth.sdk.protobuf";
option go_package = "transaction_pb2";

message TransactionHeader {
    // Public key for the client who added this transaction to a batch
    string batcher_public_key = 1;

    // A list of transaction signatures that describe the transactions that
    // must be processed before this transaction can be valid
    repeated string dependencies = 2;

    // The family name correlates to the transaction processor's family name
    // that this transaction can be processed on, for example 'intkey'
    string family_name = 3;

    // The family version correlates to the transaction processor's family
    // version that this transaction can be processed on, for example "1.0"
    string family_version = 4;

    // A list of addresses that are given to the context manager and control
    // what addresses the transaction processor is allowed to read from.
    repeated string inputs = 5;

    // A random string that provides uniqueness for transactions with
    // otherwise identical fields.
    string nonce = 6;

    // A list of addresses that are given to the context manager and control
    // what addresses the transaction processor is allowed to write to.
    repeated string outputs = 7;

    //The sha512 hash of the encoded payload
    string payload_sha512 = 9;

    // Public key for the client that signed the TransactionHeader
    string signer_public_key = 10;
}

message Transaction {
    // The serialized version of the TransactionHeader
    bytes header = 1;

    // The signature derived from signing the header
    string header_signature = 2;

    // The payload is the encoded family specific information of the
    // transaction. Example cbor({'Verb': verb, 'Name': name,'Value': value})
    bytes payload = 3;
}

// A simple list of transactions that needs to be serialized before
```

(continues on next page)

(continued from previous page)

```
// it can be transmitted to a batcher.
message TransactionList {
    repeated Transaction transactions = 1;
}
```

Header, Signature, and Public Keys

The Transaction header field is a serialized version of a TransactionHeader. The header is signed by the signer's private key (not sent with the transaction) and the resulting signature is stored in header_signature. The header is present in the serialized form so that the exact bytes can be verified against the signature upon receipt of the Transaction.

The verification process verifies that the key in signer_public_key signed the header bytes resulting in header_signature.

The batcher_public_key field must match the public key used to sign the batch in which this transaction is contained.

The resulting serialized document is signed with the transactor's private ECDSA key using the secp256k1 curve.

The validator expects a 64 byte "compact" signature. This is a concatenation of the R and S fields of the signature. Some libraries will include an additional header byte, recovery ID field, or provide DER encoded signatures. Sawtooth will reject the signature if it is anything other than 64 bytes.

Note: The original header bytes as constructed from the sender are used for verification of the signature. It is not considered good practice to de-serialize the header (for example, to a Python object) and then re-serialize the header with the intent to produce the same byte sequence as the original. Serialization can be sensitive to programming language or library, and any deviation would produce a sequence that would not match the signature; thus, best practice is to always use the original header bytes for verification.

Transaction Family

In Hyperledger Sawtooth, the set of possible transactions are defined by an extensible system called transaction families. Defining and implementing a new transaction family adds to the taxonomy of available transactions which can be applied. For example, in the language-specific tutorials that show you how to write your own transaction family (see the *Application Developer's Guide*), we define a transaction family called "xo" which defines a set of transactions for playing tic-tac-toe.

In addition to the name of the transaction family (family_name), each transaction specifies a family version string (family_version). The version string enables upgrading a transaction family while coordinating the nodes in the network to upgrade.

Dependencies and Input/Output Addresses

Transactions can depend upon other transactions, which is to say a dependent transaction cannot be applied prior to the transaction upon which it depends.

The dependencies field of a transaction allows explicitly specifying the transactions which must be applied prior to the current transaction. Explicit dependencies are useful in situations where transactions have dependencies but can not be placed in the same batch (for example, if the transactions are submitted at different times).

To assist in parallel scheduling operations, the inputs and outputs fields of a transaction contain state addresses. The scheduler determines the implicit dependencies between transactions based on interaction with state. The addresses may be fully qualified leaf-node addresses or partial prefix addresses. Input addresses are read from the state and output addresses are written to state. While they are specified by the client, input and output declarations on the transaction are enforced during transaction execution. Partial addresses work as wildcards and allow transactions to specify parts of the tree instead of just leaf nodes.

Payload

The payload is used during transaction execution as a way to convey the change which should be applied to state. Only the transaction family processing the transaction will deserialize the payload; to all other components of the system, payload is just a sequence of bytes.

The payload_sha512 field contains a SHA-512 hash of the payload bytes. As part of the header, payload_sha512 is signed and later verified, while the payload field is not. To verify the payload field matches the header, a SHA-512 of the payload field can be compared to payload_sha512.

Nonce

The nonce field contains a random string generated by the client. With the nonce present, if two transactions otherwise contain the same fields, the nonce ensures they will generate different header signatures.

2.2.2 Batch Data Structure

Batches are also serialized using Protocol Buffers. They consist of two message types:

Listing 2: File: protos/batch.proto

```
// Copyright 2016 Intel Corporation
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
// -----------------------------------------------------------------------------

syntax = "proto3";

option java_multiple_files = true;
option java_package = "sawtooth.sdk.protobuf";
option go_package = "batch_pb2";

import "transaction.proto";

message BatchHeader {
    // Public key for the client that signed the BatchHeader
    string signer_public_key = 1;

    // List of transaction.header_signatures that match the order of
    // transactions required for the batch
```

(continues on next page)

(continued from previous page)

```
    repeated string transaction_ids = 2;
}

message Batch {
    // The serialized version of the BlockHeader
    bytes header = 1;

    // The signature derived from signing the header
    string header_signature = 2;

    // A list of the transactions that match the list of
    // transaction_ids listed in the batch header
    repeated Transaction transactions = 3;

    // A debugging flag which indicates this batch should be traced through the
    // system, resulting in a higher level of debugging output.
    bool trace = 4;
}

message BatchList {
    repeated Batch batches = 1;
}
```

Header, Signature, and Public Keys

Following the pattern presented in Transaction, the Batch header field is a serialized version of a BatchHeader. The header is signed by the signer's private key (not sent with the batch) and the resulting signature is stored in header_signature. The header is present in the serialized form so that the exact bytes can be verified against the signature upon receipt of the Batch.

The resulting serialized document is signed with the transactor's private ECDSA key using the secp256k1 curve.

The validator expects a 64 byte "compact" signature. This is a concatenation of the R and S fields of the signature. Some libraries will include an additional header byte, recovery ID field, or provide DER encoded signatures. Sawtooth will reject the signature if it is anything other than 64 bytes.

Transactions

The transactions field contains a list of Transactions which make up the batch. Transactions are applied in the order listed. The transaction_ids field contains a list of Transaction header_signatures and must be the same order as the transactions field.

2.2.3 Why Batches?

As we have stated above, a batch is the atomic unit of change in the system. If a batch has been applied, all transactions will have been applied in the order contained within the batch. If a batch has not been applied (maybe because one of the transactions is invalid), then none of the transactions will be applied.

This greatly simplifies dependency management from a client perspective, since transactions within a batch do not need explicit dependencies to be declared between them. As a result, the usefulness of explicit dependencies (contained in the dependencies field on a Transaction) are constrained to dependencies where the transactions cannot be placed in the same batch.

Batches solve an important problem which cannot be solved with explicit dependencies. Suppose we have transactions A, B, and C and that the desired behavior is A, B, C be applied in that order, and if any of them are invalid, none of them should be applied. If we attempted to solve this using only dependencies, we might attempt a relationship between them such as: C depends on B, B depends on A, and A depends on C. However, the dependencies field cannot be used to represent this relationship, since dependencies enforce order and the above is cyclic (and thus cannot be ordered).

Transactions from multiple transaction families can also be batched together, which further encourages reuse of transaction families. For example, transactions for a configuration or identity transaction family could be batched with application-specific transactions.

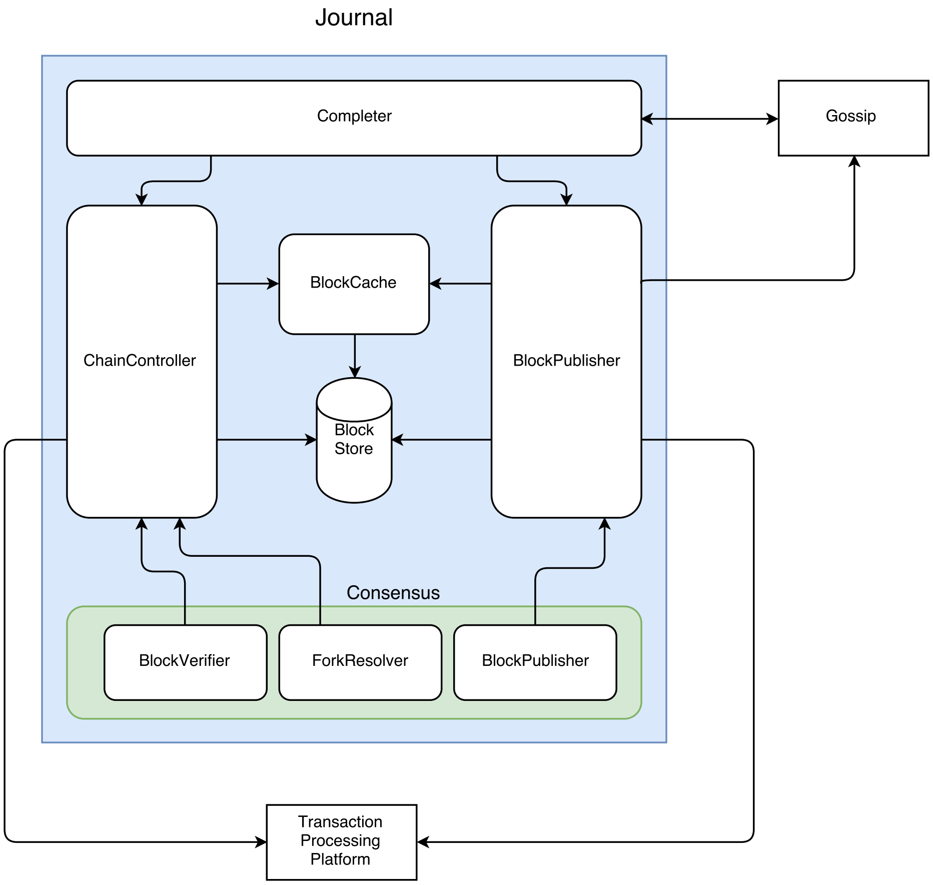
Transactions and batches can also be signed by different keys. For example, a browser application can sign the transaction and a server-side component can add transactions and create the batch and sign the batch. This enables interesting application patterns, including aggregation of transactions from multiple transactors into an atomic operation (the batch).

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_public_key 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.

2.3 Journal

The Journal is responsible for maintaining and extending the blockchain for the validator. This responsibility involves validating candidate blocks, evaluating valid blocks to determine if they are the correct chain head, and generating new blocks to extend the chain.

The Journal is the consumer of Blocks and Batches that arrive at the validator. These Blocks and Batches arrive via interconnect, either through the gossip protocol or the REST API. The newly-arrived Blocks and Batches are sent to the Journal, which routes them internally.



The Journal divides up the processing of Blocks and Batches to different pipelines. Both objects are delivered initially to the Completer, which guarantees that all dependencies for the Blocks and Batches have been satisfied and delivered downstream. Completed Blocks are delivered to the Chain controller for validation and fork resolution. Completed Batches are delivered the BlockPublisher for validation and inclusion in a Block.

The Journal is designed to be asynchronous, allowing incoming blocks to be processed in parallel by the ChainController, as well as allowing the BlockPublisher to proceed with claiming blocks even when the incoming block rate is high.

It is also flexible enough to accept different consensus algorithms. The Journal implements a consensus interface that defines the entry points and responsibilities of a consensus algorithm.

2.3.1 The BlockStore

The BlockStore contains all the blocks in the current blockchain - that is, the list of blocks from the current chain head back to the Genesis blocks. Blocks from forks are not included in the BlockStore. The BlockStore also includes a reference to the head of the current chain. It is expected to be coherent at all times, and an error in the BlockStore is considered a non-recoverable error for the validator. Such critical errors would include missing blocks, bad indexes, missing chain reference, incomplete blocks or invalid blocks in the store. The BlockStore provides an atomic means to update the store when the current fork is changed (the chain head is updated).

The BlockStore is a persistent on-disk store of all Blocks in the current chain. When the validator is started, the contents of the BlockStore is trusted to be the current state of the world.

All blocks stored here are formally complete. The BlockStore allows blocks to be accessed via Block ID. Blocks can also be accessed via Batch ID, Transaction ID, or block number; for example, `get_block_by_batch_id`, `get_block_by_transaction_id`, `get_batch_by_transaction`, or `get_block_by_number`.

The BlockStore maintains internal mappings of Transaction-to-Block and Batch-to-Block. These may be rebuilt if missing or corrupt. This rebuild should be done during startup, and not during the course of normal operation. These mappings should be stored in a format that is cached to disk, so they are not required to be held in memory at all times. As the blockchain grows, these will become quite large.

The BlockStore provides an atomic method for updating the current head of the chain. In order for the BlockStore to switch forks, it is provided with a list of blocks in the new chain to commit, and a list of blocks in the old chain to decommit. These lists are the blocks in each fork back to the common root.

2.3.2 The BlockCache

The Block Cache holds the working set of blocks for the validator and tracks the processing state. This processing state is tracked as valid, invalid, or unknown. Valid blocks are blocks that have been proven to be valid by the ChainController. Invalid blocks are blocks that failed validation or have an invalid block as a predecessor. Unknown are blocks that have not yet completed validation, usually having just arrived from the Completer.

The BlockCache is an in-memory construct. It is rebuilt by demand when the system is started.

If a block is not present in the BlockCache, it will look in the BlockStore for the block. If it is not found or the lookup fails, the block is unknown to the system. If the block is found in the BlockStore it is loaded into the BlockCache and marked as valid. All blocks in the BlockStore are considered valid.

The BlockCache keeps blocks that are currently relevant, tracked by the last time the block was accessed. Periodically, the blocks that have not been accessed recently are purged from the block cache, but only if none of the other blocks in the BlockCache reference those blocks as predecessors.

2.3.3 The Completer

The Completer is responsible for making sure Blocks and Batches are complete before they are delivered. Blocks are considered formally complete once all of their predecessors have been delivered to the ChainController and their batches field contains all the Batches specified in the BlockHeader's batch_ids list. The batches field is also expected to be in the same order as the batch_ids. Once Blocks are formally complete they are delivered to the ChainController for validation.

Batches are considered complete once all of its dependent transactions exist in the current chain or have been delivered to the BlockPublisher.

All Blocks and Batches will have a timeout for being completed. After the initial request for the missing dependencies is sent, if the response is not received within the specified time window, they are dropped.

If you have a new block of unknown validity, you must ensure that its predecessors have been delivered to the journal. If a predecessor is not delivered on request to the journal in a reasonable amount of time, the new block cannot be validated.

Consider the case where you have the chain A->B->C :

If C arrives and B is not in the BlockCache, the validator will request B. If the request for B times out, the C block is dropped.

If later on D arrives with predecessor C, of chain A->B->C->D, the Completer will request C from the network and once C arrives, then will request B again. If B arrives this time, then the new chain will be delivered to the ChainController, where they will be check for validity and considered for becoming the block head by the ChainController.

2.3.4 The Consensus Interface

In the spirit of configurability, the Journal supports *dynamic consensus algorithms* that can be changed via the Settings transaction family. The initial selection of a consensus algorithm is set for the chain in the genesis block during genesis (described below). This may be changed during the course of a chain's lifetime. The Journal and its consensus interface support dynamic consensus for probabilistic finality algorithms like Proof of Work, as well as algorithms with absolute finality like PBFT.

The Consensus algorithm services to the journal are divided into three distinct interfaces that have specific lifetimes and access to information.

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

Consensus algorithm implementations in Sawtooth must implement all of the consensus interfaces. Each of these objects are provided read-only access to the BlockCache and GlobalState.

Consensus.BlockPublisher

An implementation of the interface Consensus.BlockPublisher is used by the BlockPublisher to create new candidate blocks to extend the chain. The Consensus.BlockPublisher is provided access to a read-only view of global state, a read-only view of the BlockStore, and an interface to publish batches.

Three events are called on the Consensus.BlockPublisher,

1. initialize_block - The BlockHeader is provided for the candidate block. This is called immediately after the block_header is initialized and allows for validation of the consensus algorithm's internal state, checks if the header is correct, checks if according to the consensus rules a block could be published, and checks if any initialization of the block_header is required. If this function fails no candidate block is created and the BlockPublisher will periodically attempt to create new blocks.
2. check_publish_block - Periodically, polling is done to check if the block can be published. In the case of PoET, this is a check to see if the wait time has expired, but could be on any other criteria the consensus algorithm has for determining if it is time to publish a block. When this returns true the BlockPublisher will proceed in creating the block.
3. finalize_block - Once check_publish_block has confirmed it is time to publish a block, the block header is considered complete, except for the consensus information. The BlockPublisher calls finalize_block with the completed block_header allowing the consensus field to be filled out. Afterwards, the BlockPublisher signs the block and broadcasts it to the network.

This implementation needs to take special care to handle the genesis block correctly. During genesis operation, the Consensus.BlockPublisher will be called to initialize and finalize a block so that it can be published on the chain (see below).

Consensus.BlockVerifier

The Consensus.BlockVerifier implementation provides Block verification services to the BlockValidator. This gives the consensus algorithm an opportunity to check whether the candidate block was published following the consensus rules.

Consensus.ForkResolver

The consensus algorithm is responsible for fork resolution on the system. Depending on the consensus algorithm, the determination of the valid block to become the chain head will differ. In a Bitcoin Proof of Work consensus, this will be the longest chain, whereas PoET uses the measure of aggregate local mean (a measure of the total amount of time spent waiting) to determine the valid fork. Consensus algorithms with finality, such as PBFT, will only ever produce blocks that extend the current head. These algorithms will never have forks to resolve. The ForkResolver for these algorithms with finality will always select the new block that extends the current head.

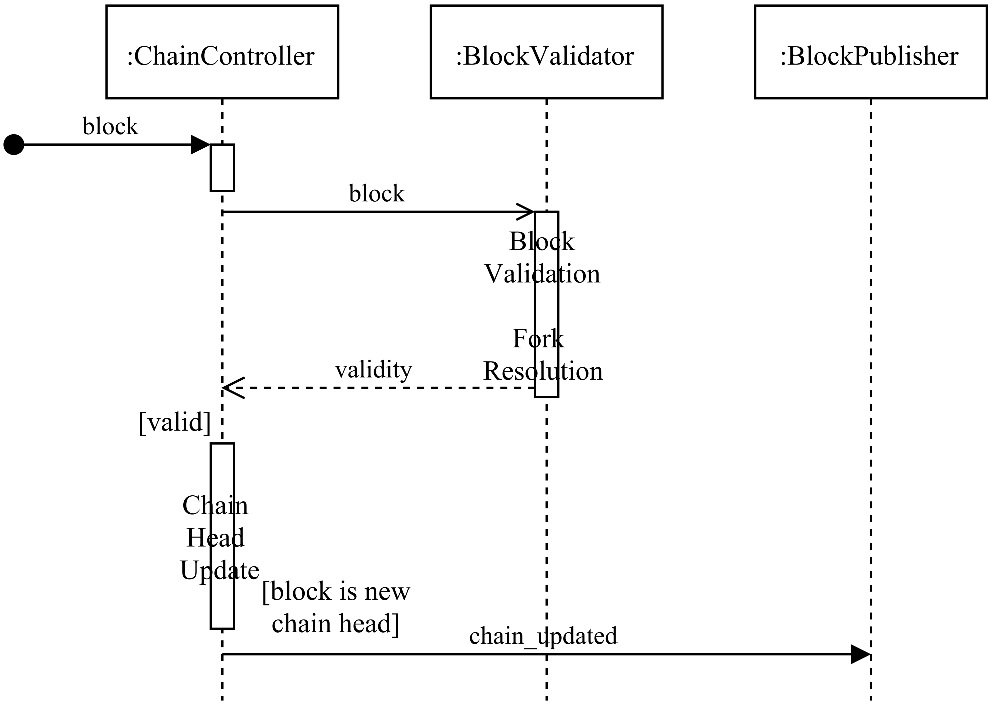
2.3.5 The ChainController

The ChainController is responsible for determining which chain the validator is currently on and coordinating any change-of-chain activities that need to happen.

The ChainController is designed to be able to handle multiple block validation activities simultaneously. For instance, if multiple forks form on the network, the ChainController can process blocks from all of the competing forks simultaneously. This is advantageous as it allows progress to be made even when there are several deep forks competing. The current chain can also be advanced while a deep fork is being evaluated. This was implemented for cases that could happen if a group of validators lost connectivity with the network and later rejoined.

Note: Currently, the thread pool is set to 1, so only one Block is validated at a time.

Here is the basic flow of the ChainController as a single block is processed.



When a block arrives, the ChainController creates a BlockValidator and dispatches it to a thread pool for execution. Once the BlockValidator has completed, it will callback to the ChainController indicating whether the new block should be the chain head. This indication falls into 3 cases:

1. The chain head has been updated since the BlockValidator was created. In this case a new BlockValidator is created and dispatched to redo the fork resolution.

2. The new Block should become the chain head. In this case the chain head is updated to be the new block.
3. The new Block should not become the chain head. This could be because the new Block is part of a chain that has an invalid block in it, or it is a member of a shorter or less desirable fork as determined by consensus.

The Chain Controller synchronizes chain head updates such that only one BlockValidator result can be processed at a time. This is to prevent the race condition of multiple fork resolution processes attempting to update the chain head at the same time.

Chain Head Update

When the chain needs to be updated, the ChainController does an update of the ChainHead using the BlockStore, providing it with the list of commit blocks that are in the new fork and a list of decommit blocks that are in the BlockStore, which must be removed. After the BlockStore is updated, the Block Publisher is notified that there is a new ChainHead.

Delayed Block Processing

While the ChainController does Block validation in parallel, there are cases where the ChainController will serialize Block validation. These cases are when a Block is received and any of its predecessors are still being validated. In this case the validation of the predecessor is completed before the new block is scheduled. This is done to avoid redoing the validation work of the predecessor Block, since the predecessor must be validated prior to the new Block, the delay is inconsequential to the outcome.

The BlockValidator

The BlockValidator is a subcomponent of the ChainController that is responsible for Block validation and fork resolution. When the BlockValidator is instantiated, it is given the candidate Block to validate and the current chain head.

During processing, if a Block is marked as invalid it is discarded, never to be considered again. The only way to have the Block reconsidered is by flushing the BlockCache, which can be done by restarting the validator.

The BlockValidator has three stages of evaluation.

1. Determine the common root of the fork (ForkRoot). This is done by walking the chain back from the candidate and the chain head until a common block is found. The Root can be the ChainHead in the case that the Candidate is advancing the existing chain. The only case that the ForkRoot will not be found is if the Candidate is from another Genesis. If this is the case, the Candidate and all of its predecessors are marked as Invalid and discarded. During this step, an ordered list of both chains is built back to the ForkRoot.
2. The Candidate chain is validated. This process walks forward from the ForkRoot and applies block validation rules (described below) to each Block successively. If any block fails validation, it and all of its successors are marked as Invalid (Valid Blocks are defined as having Valid predecessor(s)). Once the Candidate is successfully Validated and marked as Valid, the Candidate is ready for Fork Resolution.
3. Fork resolution requires a determination to be made if the Candidate should replace the ChainHead and is deferred entirely to the consensus implementation. Once the Consensus determines if the block is the new ChainHead, the answer is returned to the ChainController, which updates the BlockStore. If it is not the new ChainHead, the Candidate is dropped. Additionally, if the Candidate is to become the ChainHead, the list of transactions committed in the new chain back to the common root is computed and the same list is computed on the current chain. This information helps the BlockPublisher update its pending batch list when the chain is updated.

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.

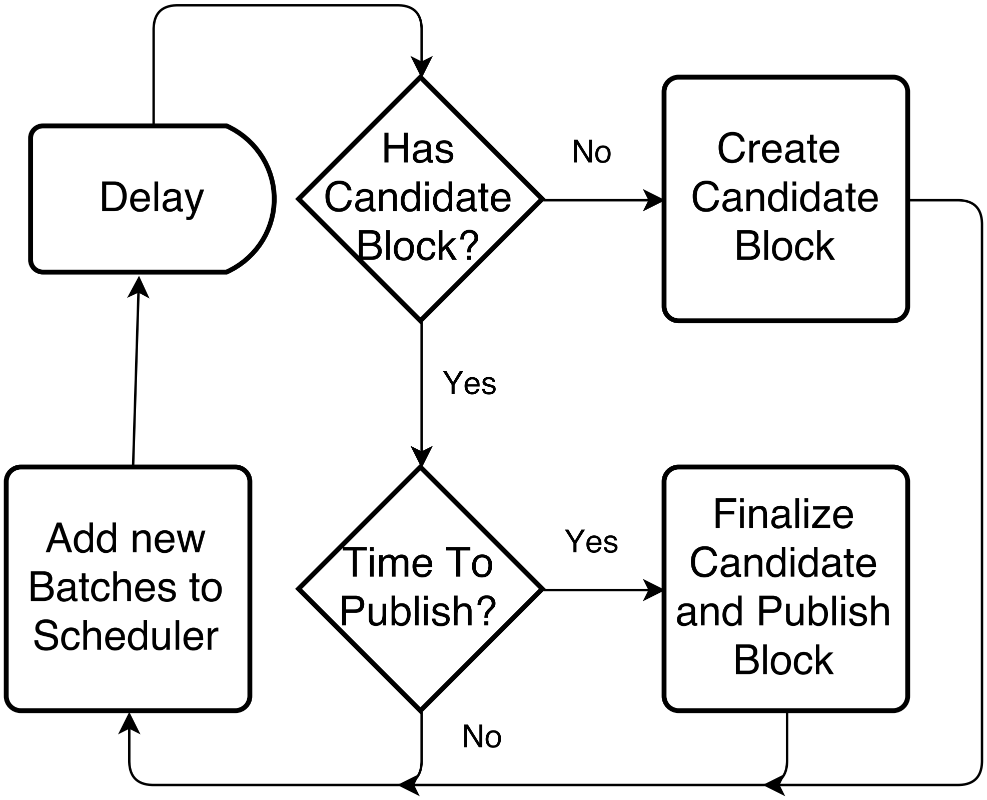
1. **Transaction Permissioning** - On-chain transaction permissions are checked to see who is allowed to submit transactions and batches.
2. **On-chain Block Validation Rules** - The on-chain block validation rules are checked to ensure that the Block doesn't invalidate any of the rules stored at `sawtooth.validator.block_validation_rules`.
3. **Batches Validation** - All of the Batches in the block are sent in order to a Transaction Scheduler for validation. If any Batches fail validation, this block is marked as invalid. Note: Batch and Signature verification is done on receipt of the Batch prior to it being routed to the Journal. The batches are checked for the following:
   - No duplicate Batches
   - No duplicate Transactions
   - Valid Transaction dependencies
   - Successful Batch Execution
4. **Consensus Verification** - The Consensus instance is given to the Block for verification. Consensus block verification is done by the consensus algorithm using its own rules.
5. **State Hash Check** - The StateRootHash generated by validating the block is checked against the StateRootHash (state_root_hash field in the BlockHeader) on the block. They must match for the block to be valid.

If the block is computed to be valid, then StateRootHash is committed to the store.

2.3.6 The BlockPublisher

The BlockPublisher is responsible for creating candidate blocks to extend the current chain. The BlockPublisher does all of the housekeeping work around creating a block but takes direction from the consensus algorithm for when to create a block and when to publish a block.

The BlockPublisher follows this logic flow:



At each processing stage, the consensus algorithm has a chance to inspect and confirm the validity of the block.

During CreateBlock, an instance of Consensus.BlockPublisher is created that is responsible for guiding the creation of this candidate block. Also, a TransactionScheduler is created and all of the pending Batches are submitted to it.

A delay is employed in the checking loop to ensure that there is time for the batch processing to occur.

2.3.7 Genesis Operation

The Journal supports Genesis operation. This is the action of creating a root of the chain (the Genesis block) when the block store is empty. This operation is necessary for bootstrapping a validator network with the desired consensus model, any deployment-specific configuration settings, as well as any genesis-time transactions for an application's Transaction Family.

Genesis Batch Creation

The CLI tool produces batches in a file, which will be consumed by the validator on startup (when starting with an empty chain).

The file contains a protobuf-encoded list of batches:

Listing 3: File: sawtooth-core/protos/genesis.proto

```
message GenesisData {
    repeated Batch batches = 1;
}
```

The tool should take multiple input batch collections, and combine them together into the single list of batches contained in GenesisData. This allows independent tools or transaction families to include their own batches, without needing to know anything about the genesis process.

The first implementation assumes that the order of the input batches have implied dependencies, with each batch being implicitly dependent on the previous. Any dependencies should be verified when the final set of batches is produced. This would be enforced by the use of strict ordering of the batches during execution time. Future implementations may provide a way to verify dependencies across input batches.

Transaction family authors who need to provide batches that will be included, need to provide their own tool to produce GenesisData, with the batches they require for the process. Each individual tool may manage their batch and transaction dependencies explicitly within the context of their specific genesis batches.

Example

The following example configures the validator to use PoET consensus and specifies the appropriate settings:

```
sawset proposal create \
  -k <signing-key-file> \
  -o sawset.batch \
  sawtooth.consensus.algorithm=poet \
  sawtooth.poet.initial_wait_timer=x \
  sawtooth.poet.target_wait_time=x \
  sawtooth.poet.population_estimate_sample_size=x
  sawadm genesis \
  sawset.batch
```

A genesis.batch file will written to the validator’s data directory.

Block Creation

On startup, the validator would use the resulting genesis.batch file to produce a genesis block under the following conditions:

- The genesis.batch file exists
- There is no block specified as the chain head

If either of these conditions is not met, the validator halts operation.

The validator will load the batches from the file into the pending queue. It will then produce the genesis block through the standard process with the following modifications.

First, the execution of the batches will be strictly in the order they have been provided. The Executor will not attempt to reorder them, or drop failed transactions. Any failure of a transaction in genesis.batch will fail to produce the genesis block, and the validator will treat this as a fatal error.

Second, it will use a genesis consensus, to determine block validity. At the start of the genesis block creation process, state (the Merkle-Radix tree) will be empty. Given that the consensus mechanism is specified by a configuration setting in the state, this will return None. As a result, the genesis consensus mechanism will be used. This will produce a block with an empty consensus field.

In addition to the genesis block, the blockchain ID (that is, the signature of the genesis block) is written to the file `block-chain-id` in the validator's data directory.

Part of the production of the genesis block will require the configuration of the consensus mechanism. The second block will then use the configured consensus model, which will need to know how to initialize the consensus field from an empty one. In future cases, transitions between consensus models may be possible, as long as they know how to read the consensus field of the previous block.

To complete the process, all necessary transaction processors must be running. A minimum requirement is the Sawtooth Settings transaction processor, `settings-tp`.

2.4 Transaction Scheduling

Sawtooth supports both serial and parallel scheduling of transactions. The scheduler type is specified via a command line argument or as an option in the validator's configuration file when the validator process is started. Both schedulers result in the same deterministic results and are completely interchangeable.

The parallel processing of transactions provides a performance improvement for even fast transaction workloads by reducing overall latency effects which occur when transaction execution is performed serially. When transactions are of non-uniform duration (such as may happen with more realistic complex workloads), the performance benefit is especially magnified when faster transactions outnumber slower transactions.

Transaction scheduling and execution in Sawtooth correctly and efficiently handles transactions which modify the same state addresses, including transactions within the same block. Even at the batch level, transactions can modify the same state addresses. Naive distributed ledger implementations may not allow overlapping state modifications within a block, severely limiting performance of such transactions to one-transaction-per-block, but Sawtooth has no such block-level restriction. Instead, state is incremental per transaction execution. This prevents double spends while allowing multiple transactions which alter the same state values to appear in a single block. Of course, in cases where these types of block-level restrictions are desired, transaction families may implement the appropriate business logic.

2.4.1 Scheduling within the Validator

The sawtooth-validator process has two major components which use schedulers to calculate state changes and the resulting Merkle hashes based on transaction processing: the Chain Controller and the Block Publisher. The Chain Controller and Block Publisher pass a scheduler to the Executor. While the validator contains only a single Chain Controller, a single Block Publisher, and a single Executor, there are numerous instances of schedulers which are dynamically created as needed.

Chain Controller

The Chain Controller is responsible for maintaining the current chain head (a pointer to the last block in the current chain). Processing is block-based; when it receives a candidate block (over the network or from the Block Publisher), it determines whether the chain head should be updated to point to that candidate block. The Chain Controller creates a scheduler to calculate new state with a related Merkle hash for the block being published. The Merkle hash is compared to the state root contained in the block header. If they match, the block is valid from a transaction execution and state standpoint. The Chain Controller uses this information in combination with consensus information to determine whether to update the current chain head.

Block Publisher

The Block Publisher is responsible for creating new candidate blocks. As batches are received by the validator (from clients or other validator nodes), they are added to the Block Publisher's pending queue. Only valid transactions will be added to the next candidate block. For timeliness, batches are added to a scheduler as they are added to the pending queue; thus, transactions are processed incrementally as they are received.

When the pending queue changes significantly, such as when the chain head has been updated by the Chain Controller, the Block Publisher cancels the current scheduler and creates a new scheduler.

Executor

The Executor is responsible for the execution of transactions by sending them to transaction processors. The overall flow for each transaction is:

- The Executor obtains the next transaction and initial context from the scheduler
- The Executor obtains a new context for the transaction from the Context Manager by providing the initial context (contexts are chained together)
- The Executor sends the transaction and a context reference to the transaction processor
- The transaction processor updates the context's state via context manager calls
- The transaction processor notifies the Executor that the transaction is complete
- The Executor updates the scheduler with the transaction's result with the updated context

In the case of serial scheduling, step (1) simply blocks until step (6) from the previous transaction has completed. For the parallel scheduler, step (1) blocks until a transaction exists which can be executed because it's dependencies have been satisfied, with steps (2) through (6) happening in parallel for each transaction being executed.

2.4.2 Iterative Scheduling

Each time the executor requests the next transaction, the scheduler calculates the next transaction dynamically based on knowledge of the transaction dependency graph and previously executed transactions within this schedule.

Serial Scheduler

For the serial scheduler, the dependency graph is straightforward; each transaction is dependent on the one before it. The next transaction is released only when the scheduler has received the execution results from the transaction before it.

Parallel Scheduler

As batches are added to the parallel scheduler, predecessor transactions are calculated for each transaction in the batch. A predecessor transaction is a transaction which must be fully executed prior to executing the transaction for which it is a predecessor.

Each transaction has a list of inputs and outputs; these are address declarations fields in the transaction's header and are filled in by the client when the transaction is created. Inputs and outputs specify which locations in state are accessed or modified by the transaction. Predecessor transactions are determined using these inputs/outputs declarations.

Note: It is possible for poorly written clients to impact parallelism by providing overly broad inputs/outputs declarations. Transaction processor implementations can enforce specific inputs/outputs requirements to provide an incentive for correct client behavior.

The parallel scheduler calculates predecessors using a Merkle-Radix tree with nodes addressable by state addresses or namespaces. This tree is called the predecessor tree. Input declarations are considered reads, with output declarations considered writes. By keeping track of readers and writers within nodes of the tree, predecessors for a transaction can be quickly determined.

Unlike the serial scheduler, the order in which transactions will be returned to the Executor is not predetermined. The parallel scheduler is careful about which transactions are returned; only transactions with do not have state conflicts will be executed in parallel. When the Executor asks for the next transaction, the scheduler inspects the list of unscheduled transactions; the first in the list for which all predecessors have finished executed will be be returned. If none are found, the scheduler will block and re-check after a transaction has finished being executed.

2.5 REST API

Hyperledger Sawtooth provides a pragmatic RESTish API for clients to interact with a validator using common HTTP/JSON standards. It is an entirely separate process, which once running, allows transactions to be submitted and blocks to be read with a common language-neutral interface. As the validator is redesigned and improved the REST API will grow with it, providing a consistent interface that meets the needs of application developers into the future.

With that focus on application developers, the REST API treats the validator mostly as a black box, submitting transactions and fetching the results. It is not the tool for all validator communication. For example, it is not used by Transaction Processors to communicate with a validator, or by one validator to talk to other validators. For these and similar use cases, there is a more efficient and robust, if somewhat more complicated, ZMQ/Protobuf interface.

2.5.1 Open API Specification

The REST API is comprehensively documented using the OpenAPI specification (fka Swagger), formatted as a YAML file. This documentation provides a single source of truth that completely documents every implemented aspect of the API, is both human and machine readable, and can be compiled by a variety of toolsets (including for this Sphinx-based document). The spec file itself can be found here.

2.5.2 HTTP Status Codes

In order to improve clarity and ease parsing, the REST API supports a limited number of common HTTP status codes. Further granularity for parsing errors is provided by specific *Error Responses* in the JSON response envelope itself (see below).

Code	Title	Description
200	OK	The requested resources were successfully fetched. They are included in the `"data"` property of the response envelope.
201	Created	The POSTed resources were submitted and created/committed successfully. A `"link"` is included to the resources.
202	Accepted	The POSTed resources were submitted to the validator, but are not yet committed. A `"link"` to check the *status* of the submitted resources is included. If told to wait for commit, but timed out, the status at the moment of timing out is also included in the `"data"` property.
400	Bad Request	Something in the client's request was malformed, and the request could not be completed.
404	Not Found	The request was well-formed, but there is no resource with the identifier specified.
500	Internal Server Error	Something is broken in the REST API or the validator. If consistently reproducible, a bug report should be submitted.
503	Service Unavailable	Indicates the REST API is unable to contact the validator.

2.5.3 Data Envelope

The REST API uses a JSON envelope to send metadata back to clients in a way that is simple to parse and easily customized. All successful requests will return data in an envelope with at least one of four possible properties:

- **data** - contains the actual resource or resources being fetched
- **head** - id of the head block of the chain the resource was fetched from, this is particularly useful to know if an explicit *head* was not set in the original request
- **link** - a link to the resources fetched with both *head* and paging parameters explicitly set, will always return the same resources
- **paging** - information about how the resources were paginated, and how further pages can be fetched (see below)

Example response envelope:

```
{
  "data": [{"fetched": "resources"}],
  "head": "65cd...47dd",
  "link": "http://rest.api.domain/state?head=65cd...47dd"
}
```

Pagination

All endpoints that return lists of resources will automatically paginate results, limited by default to 1000 resources. In order to specify what range of resources to return, these endpoints take `count` and either `min` or `max` query parameters. They specify the total number of items to include, as well as what the first or last item in the list should be. For convenience, both *min* and *max* may refer to either an index or a resource id.

Example paged request URL:

```
http://rest.api.domain/blocks?count=100&min=200
```

Within the `"paging"` property of the response body there will be one or more of these four values:

- **start_index** - index of the first item in the fetched list
- **total_count** - total number of resources available
- **previous** - URL for the previous page, if any
- **next** - URL for the next page, if any

Example paging response:

```
{
  "data": [{"fetched": "resources"}],
  "paging": {
    "start_index": 200,
    "total_count": 54321,
    "previous": "http://rest.api.domain/state?head=65cd...47dd&count=100&min=100",
    "next": "http://rest.api.domain/state?head=65cd...47dd&count=100&min=300"
  }
}
```

Errors

If something goes wrong while processing a request, the REST API will send back a response envelope with only one property: `"error"`. That error will contain three values which explain the problem that occurred:

- **code** - machine-parsable code specific to this particular error
- **title** - short human-readable headline for the error
- **message** - longer more detailed explanation of what went wrong

Example error response:

```
{
  "error": {
    "code": 30,
    "title": "Submitted Batches Invalid",
    "message": "The submitted BatchList is invalid. It was poorly formed, or has an␣
↪invalid signature."
  }
}
```

Note: While the title or message of an error may change or be reworded over time, **the code is fixed**, and will always refer to the same error.

2.5.4 Query Parameters

Many routes support query parameters to help specify how a request to the validator should be formed. Not every endpoint supports every query, and some endpoints have their own parameters specific to just to them. Any queries specific to a single endpoint are not listed here.

head	The id of the block to use as the chain head. This is particularly useful to request older versions of state *(defaults to the latest chain head)*.
count	For paging, this item specifies the number of resources to fetch *(defaults to 1000)*.
min	For paging, specifies the id or index of the first resource to fetch *(defaults to 0)*.
max	For paging, specifies the id or index of the last resource to fetch. It would be used instead of *min*, not in the same query.
sort	For endpoints that fetch lists of resources, specifies a key or keys to sort the list by. These key sorts can be modified with a few simple rules: nested keys can be dot-notated; *header.* may be omitted in the case of nested header keys; appending *.length* sorts by the length of the property; a minus-sign specifies descending order; multiple keys can be used if comma-separated. For example: *?sort=header.signer_public_key,-transaction_ids.length*
wait	For submission endpoints, instructs the REST API to wait until batches have been committed to the blockchain before responding to the client. Can be set to a positive integer to specify a timeout in seconds, or without any value to use the REST API's internal timeout.

2.5.5 Endpoints

The endpoints include RESTful references to resources stored in the Sawtooth ledger that clients might be interested in, like blocks and transactions, as well as RESTish metadata, like batch status.

Resource Endpoints

In order to fetch resources stored on chain or in the validator's state, various resource routes are provided. As is typical with RESTful APIs, a `GET` request fetches one or many resources, depending on whether or not a particular resource identifier was specified (i.e., `/resources` vs `/resources/{resource-identifier}`).

- **/blocks** - the actual blocks currently in the blockchain, referenced by id (aka `header_signature`)
- **/batches** - the batches stored on the blockchain, referenced by id
- **/transactions** - the transactions stored on the blockchain, referenced by id
- **/state** - the ledger state, stored on the Merkle-Radix tree, referenced by leaf addresses

Submission Endpoints

In order to submit transactions to a Sawtooth validator, they *must* be wrapped in a batch. For that reason, submissions are sent to the `/batches` endpoint and only that endpoint. Due to the asynchronous nature of blockchains, there is a corresponding endpoint to check the status of submitted batches. Both requests will accept the `wait` query parameter, allowing clients to receive a response only once the batches are committed.

- **/batches** - accepts a `POST` request with a body of a binary BatchList of batches to be submitted
- **/batch_statuses** - fetches the committed status of one or more batches

  Example batch status response:

```
[
  {
    "id":
↪"89807bfc9089e37e00d87d97357de14cfbc455cd608438d426a625a30a0da9a31c406983803c4aa27e1f32a3ff617
↪",
    "status": "COMMITTED"
  },
  {
    "id":
↪"c0c2075e708c04b34903c5374f65c9352f9dc9662f187e4bab0605aba3eb697e459bfa3a61a8050c428d1347d47a1
↪",
    "status": "PENDING"
  }
]
```

2.5.6 Future Development

Stats and Status Endpoints

In order to track the performance of the validator and the blockchain generally, additional endpoints could be implemented to fetch metrics related to block processing, peer to peer communication, system status, and more. These will require significant design and development, both within the REST API, and within the core validator code itself.

Configuration

At some point it may be useful to add some configuration options to the REST API, such as:

- Modify error verbosity to change detail and security sensitivity of error messages provided (i.e. whether or not to include the stack trace)

- Enable or disable the stats and status endpoints

Authorization

The current intention is for the REST API to be a lightweight shim on top of the internal ZMQ communications. From this perspective, the API offers no authorization, simply passing through every request to the validator to be authorized with signature verification or some other strategy defined by an individual Transaction Processor.

However, that may be insufficient if the API needed to be deployed just for certain authorized clients. In that use case, the best solution would be to expand the REST API to handle the validation of *API keys*. In their most basic form, these can be validated programmatically without any need for persistent state stored on a database or elsewhere. However, more sophisticated functionality, like blacklisting particular keys that are compromised, would require some strategy for persistent storage.

Note: While the REST API does not support any sort of authorization internally, it is entirely possible to put it behind a proxy that does. See: *Using a Proxy Server to Authorize the REST API*

2.6 PoET 1.0 Specification

2.6.1 Introduction

The Proof of Elapsed Time (PoET) Consensus method offers a solution to the Byzantine Generals Problem that utilizes a “trusted execution environment” to improve on the efficiency of present solutions such as Proof-of-Work. The initial reference implementation of PoET released to Hyperledger was written for an abstract TEE to keep it flexible to any TEE implementation. This specification defines a concrete implementation for SGX. The following presentation assumes the use of Intel SGX as the trusted execution environment.

At a high-level, PoET stochastically elects individual peers to execute requests at a given target rate. Individual peers sample an exponentially distributed random variable and wait for an amount of time dictated by the sample. The peer with the smallest sample wins the election. Cheating is prevented through the use of a trusted execution environment, identity verification and blacklisting based on asymmetric key cryptography, and an additional set of election policies.

For the purpose of achieving distributed consensus efficiently, a good lottery function has several characteristics:

- Fairness: The function should distribute leader election across the broadest possible population of participants.
- Investment: The cost of controlling the leader election process should be proportional to the value gained from it.
- Verification: It should be relatively simple for all participants to verify that the leader was legitimately selected.

PoET is designed to achieve these goals using new secure CPU instructions which are becoming widely available in consumer and enterprise processors. PoET uses these features to ensure the safety and randomness of the leader election process without requiring the costly investment of power and specialized hardware inherent in most “proof” algorithms.

Sawtooth includes an implementation which simulates the secure instructions. This should make it easier for the community to work with the software but also forgoes Byzantine fault tolerance.

PoET essentially works as follows:

1. Every validator requests a wait time from an enclave (a trusted function).
2. The validator with the shortest wait time for a particular transaction block is elected the leader.

3. One function, such as "CreateTimer", creates a timer for a transaction block that is guaranteed to have been created by the enclave.

4. Another function, such as "CheckTimer", verifies that the timer was created by the enclave. If the timer has expired, this function creates an attestation that can be used to verify that validator did wait the allotted time before claiming the leadership role.

The PoET leader election algorithm meets the criteria for a good lottery algorithm. It randomly distributes leadership election across the entire population of validators with distribution that is similar to what is provided by other lottery algorithms. The probability of election is proportional to the resources contributed (in this case, resources are general purpose processors with a trusted execution environment). An attestation of execution provides information for verifying that the certificate was created within the enclave (and that the validator waited the allotted time). Further, the low cost of participation increases the likelihood that the population of validators will be large, increasing the robustness of the consensus algorithm.

2.6.2 Definitions

The following terms are used throughout the PoET spec and are defined here for reference.

Enclave A protected area in an application's address space which provides confidentiality and integrity even in the presence of privileged malware.

The term can also be used to refer to a specific enclave that has been initialized with a specific code and data.

Basename A service provider base name. In our context the service provider entity is the distributed ledger network. Each distinct network should have its own Basename and Service Provider ID (see EPID and IAS specifications).

EPID An anonymous credential system. See E. Brickell and Jiangtao Li: "Enhanced Privacy ID from Bilinear Pairing for Hardware Authentication and Attestation". IEEE International Conference on Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust. 2010.

EPID Pseudonym Pseudonym of an SGX platform used in linkable quotes. It is part of the IAS attestation response according to IAS API specifications. It is computed as a function of the service Basename (validator network in our case) and the device's EPID private key.

PPK, PSK PoET ECC public and private key created by the PoET enclave.

IAS Report Key IAS public key used to sign attestation reports as specified in the current IAS API Guide.

PSEmanifest Platform Services Enclave manifest. It is part of an SGX quote for enclaves using Platform Services like Trusted Time and Monotonic Counters.

AEP Attestation evidence payload sent to IAS (see IAS API specifications). Contains JSON encodings of the quote, an optional PSEmanifest, and an optional nonce.

AVR Attestation verification report, the response to a quote attestation request from the IAS. It is verified with the IAS Report Key. It contains a copy of the input AEP.

$WaitCertId_n$ The n-th or most recent WaitCertificate digest. We assume $n >= 0$ represents the current number of blocks in the ledger. WaitCertId is a function of the contents of the Wait Certificate. For instance the SHA256 digest of the WaitCertificate ECDSA signature.

OPK, OSK Originator ECDSA public and private key. These are the higher level ECDSA keys a validator uses to sign messages.

OPKhash SHA256 digest of OPK

blockDigest ECDSA signature with OSK of SHA256 digest of transaction block that the validator wants to commit.

localMean Estimated wait time local mean.

MCID SGX Monotonic Counter identifier.

SealKey The SGX enclave Seal Key. It is used by the SGX `sgx_seal_data()` and `sgx_unseal_data()` functions.

PoetSealKey The Poet SGX enclave Seal Key. It must be obtained through the SGX SDK `` ` ``sgx_get_key() function passing a fixed 32 byte constant as `key_id` argument.

PoET_MRENCLAVE Public MRENCLAVE (see SGX SDK documentation) value of valid PoET SGX enclave.

T_{WT} WaitTimer timeout in seconds. A validator has at most T_{WT} seconds to consume a WaitTimer, namely obtain a WaitCertificate on it after the WaitTimer itself has expired.

K Number of blocks a validator can commit before having to sign-up with a fresh PPK.

c The "sign-up delay", i.e., number of blocks a validator has to wait after sign-up before starting to participate in elections.

minDuration Minimum duration for a WaitTimer.

2.6.3 P2P PoET SGX Enclave Specifications

The P2P PoET SGX enclave uses the following data structures:

```
WaitTimer {
  double requestTime
  double duration
  byte[32] WaitCertId:sub:`n`
  double localMean
}

WaitCertificate {
  WaitTimer waitTimer
  byte[32] nonce
  byte[] blockDigest
}
```

It uses the following global variables:

```
WaitTimer activeWT # The unique active WaitTimer object
byte[64] PPK
byte[64] PSK
MCID # SGX Monotonic Counter Identifier
```

It exports the following functions:

`generateSignUpData(OPKhash)`

Returns

```
byte[64]  PPK
byte[432] report # SGX Report Data Structure
byte[256] PSEmanifest
byte[672] sealedSignUpData # (PPK, PSK, MCID) tuple encrypted with SealKey
```

Parameters

```
byte[32] OPKhash # SHA256 digest of OPK
```

Description

1. Generate fresh ECC key pair (PPK, PSK)
2. Create monotonic counter and save its identifier as MCID.
3. Use the SGX `sgx_seal_data()` function to encrypt (PPK, PSK, MCID) with SealKey (using MRENCLAVE policy) $sealedSignupData = \text{AES-GCM}_{SealKey}(PPK|PSK|MCID)$
4. Create SGX enclave report, store `SHA256(OPKhash|PPK)` in `report_data` field.
5. Get SGX PSE manifest: PSEManifest.
6. Save (PPK, PSK, MCID) as global variables within the enclave.
7. Set active WaitTimer instance activeWT to NULL.
8. Return (PPK, report, PSEmanifest, sealedSignUpData).

Note: **Implementation Note:** Normally, there is a maximum number of monotonic counters that can be created. One way to deal with this limitation is to destroy a previously created monotonic counter if this is not the first time the generateSignupData function was called.

unsealSignUpData(sealedSignUpData)

Returns

```
byte[64] PPK
```

Parameters

```
byte[672] sealedSignUpData # (PPK, PSK, MCID) tuple encrypted with SealKey
```

Description

1. Use the `sgx_unseal_data()` function to decrypt sealedSignUpData into (PPK, PSK, MCID) with SealKey (using MRENCLAVE policy).
2. Save (PPK, PSK, MCID) as global variables within the enclave.
3. Set global active WaitTimer instance activeWT to NULL.
4. Return PPK

createWaitTimer(localMean, WaitCertId_n)

Returns

```
WaitTimer waitTimer
byte[64] signature # ECDSA PSK signature of waitTimer
```

Parameters

```
double localMean # Estimated wait time local mean
byte[32] WaitCertId_n # SHA256 digest of WaitCertificate owner's ECDSA
                      # signature
```

Description

1. Increment monotonic counter MCID and store value in global variable counterValue.

2. Compute $tag = \text{AES-CMAC}_{PoetSealKey}(WaitCertId_n)$.
3. Convert lowest 64-bits of tag into double precision number in $[0, 1]$: tagd.
4. Compute $duration = minimumDuration - localMean * log(tagd)$.
5. Set requestTime equal to SGX Trusted Time value.
6. Create WaitTimer object $waitTimer = WaitTimer(requestTime, duration, WaitCertId_n, localMean)$.
7. Compute ECDSA signature of waitTimer using PSK: $signature = ECDSA_{PSK}(waitTimer)$.
8. Set global active WaitTimer instance activeWT equal to waitTimer.
9. Return (waitTimer, signature).

`createWaitCertificate(blockDigest)`

Returns

```
WaitCertificate waitCertificate
byte[64] signature # ECDSA PSK signature of waitCertificate
```

Parameters

```
byte[] blockDigest # ECDSA signature with originator private key of SHA256
                   # digest of transaction block that the validator wants
                   # to commit
```

Description

1. If activeWT is equal to NULL, exit.
2. Read monotonic counter MCID and compare its value to global variable counterValue. If values do not match, exit.
3. Read SGX Trusted time into variable currentTime. If currentTime is smaller than $waitTimer.requestTime + waitTimer.duration$, exit (the duration has not elapsed yet).
4. If currentTime is larger than $waitTimer.requestTime + waitTimer.duration + T_{WT}$, exit.
5. Generate random nonce.
6. Create WaitCertificate object $waitCertificate = WaitCertificate(waitTimer, nonce, blockDigest)$.
7. Compute ECDSA signature of waitCertificate using PSK: $signature = ECDSA_{PSK}(waitCertificate)$.
8. Set activeWT to NULL.
9. Return (waitCertificate, signature).

Sign-up Phase

A participant joins as a validator by downloading the PoET SGX enclave and a SPID certificate for the blockchain. The client side of the validator runs the following sign-up procedure:

1. Start PoET SGX enclave: ENC.
2. Generate sign-up data: $(PPK, report, PSEmanifest, sealedSignUpData) =$ ENC.generateSignUpData(OPKhash) The `report_data` (512 bits) field in the report body includes the SHA256 digest of (OPKhash | PPK).
3. Ask SGX Quoting Enclave (QE) for linkable quote on the report (using the validator network's Basename).

4. If Self Attestation is enabled in IAS API: request attestation of linkable quote and PSE manifest to IAS. The AEP sent to IAS must contain:
    - isvEnclaveQuote: base64 encoded quote
    - pseManifest: base64 encoded PSEmanifest
    - nonce: $WaitCertId_n$

    The IAS sends back a signed AVR containing a copy of the input AEP and the EPID Pseudonym.
5. If Self Attestation is enabled in IAS API: broadcast self-attested join request, (OPK, PPK, AEP, AVR) to known participants.
6. If Self Attestation is NOT enabled in IAS API: broadcast join request, (OPK, PPK, quote, PSEmanifest) to known participants.

A validator has to wait for c block to be published on the distributed ledger before participating in an election.

The server side of the validator runs the following sign-up procedure:

1. Wait for a join request.
2. Upon arrival of a join request do the verification:

    If the join request is self attested (Self Attestation is enabled in IAS API): (OPK, PPK, AEP, AVR)

    (a) Verify AVR legitimacy using IAS Report Key and therefore quote legitimacy.

    (b) Verify the `report_data` field within the quote contains the SHA256 digest of (OPKhash | PPK).

    (c) Verify the nonce in the AVR is equal to $WaitCertId_n$, namely the digest of the most recently committed block. It may be that the sender has not seen $WaitCertId_n$ yet and could be sending $WaitCertId_{n'}$ where $n' < n$. In this case the sender should be urged to updated his/her view of the ledger by appending the new blocks and retry. It could also happen that the receiving validator has not seen $WaitCertId_n$ in which case he/she should try to update his/her view of the ledger and verify again.

    (d) Verify MRENCLAVE value within quote is equal to PoET_MRENCLAVE (there could be more than one allowed value).

    (e) Verify PSE Manifest SHA256 digest in AVR is equal to SHA256 digest of PSEmanifest in AEP.

    (f) Verify basename in the quote is equal to distributed ledger Basename.

    (g) Verify attributes field in the quote has the allowed value (normally the enclave must be in initialized state and not be a debug enclave).

    If the join request is not self attested (Self Attestation is NOT enabled in IAS API): (OPK, PPK, quote, PSEmanifest)

    (a) Create AEP with quote and PSEmanifest :
    - isvEnclaveQuote: base64 encoded quote
    - pseManifest: base64 encoded PSEmanifest

    (b) Send AEP to IAS. The IAS sends back a signed AVR.

    (c) Verify received AVR attests to validity of both quote and PSEmanifest and save EPID Pseudonym.

    (d) Verify `report_data` field within the quote contains the SHA256 digest of (OPKhash | PPK).

    (e) Verify MRENCLAVE value within quote is equal to PoET_MRENCLAVE (there could be more than one allowed value).

    (f) Verify basename in the quote is equal to distributed ledger Basename.

(g) Verify attributes field in the quote has the allowed value (normally the enclave must be in initialized state and not be a debug enclave).

If the verification fails, exit.

If the verification succeeds but the SGX platform identified by the EPID Pseudonym in the quote has already signed up, ignore the join request, exit.

If the verification succeeds:

(a) Pass sign-up certificate of new participant (OPK, EPID Pseudonym, PPK, current $WaitCertId_n$ to upper layers for registration in EndPoint registry.

(b) Goto 1.

Election Phase

Assume the identifier of the most recent valid block is $WaitCertId_n$. Broadcast messages are signed by a validator with his/her PPK. To participate in the election phase a validator runs the following procedure on the client side:

1. Start the PoET SGX enclave: ENC.
2. Read the sealedSignUpData from disk and load it into enclave: ENC.unsealSignUpData$(sealedSignUpData)$
3. Call $(waitTimer, signature) = ENC.createWaitTimer(localMean, WaitCertId_n)$.
4. Wait waitTimer.duration seconds.
5. Call $(waitCertificate, signature) = ENC.createWaitCertificate(blockDigest)$.
6. If the `createWaitCertificate()` call is successful, broadcast (waitCertificate, signature, block, OPK, PPK) where block is the transaction block identified by blockDigest.

On the server side a validator waits for incoming (waitCertificate, signature, block, OPK, PPK) tuples. When one is received the following validity checks are performed:

1. Verify the PPK and OPK belong to a registered validator by checking the EndPoint registry.
2. Verify the signature is valid using sender's PPK.
3. Verify the PPK was used by sender to commit less than K blocks by checking EndPoint registry (otherwise sender needs to re-sign).
4. Verify the waitCertificate.waitTimer.localMean is correct by comparing against localMean computed locally.
5. Verify the waitCertificate.blockDigest is a valid ECDSA signature of the SHA256 hash of block using OPK.
6. Verify the sender has been winning elections according to the expected distribution (see z-test documentation).
7. Verify the sender signed up at least c committed blocks ago, i.e., respected the c block start-up delay.

A valid waitCertificate is passed to the upper ledger layer and the waitCertificate with the lowest value of waitCertificate.waitTimer.duration determines the election winner.

Revocation

Two mechanisms are put in place to blacklist validators whose EPID key has been revoked by IAS. The first one affects each validator periodically, although infrequently. The second one is an asynchronous revocation check that each validator could perform on other validators' EPID keys at any time.

1. **Periodic regeneration of PPK** a validator whose EPID key has been revoked by the IAS would not be able to obtain any valid AVR and therefore would be prevented from signing-up. Forcing validators to periodically re-sign with a fresh sign-up certificate leaves validators whose EPID keys have been revoked out of the system. Validators have to re-sign after they commit K blocks and if they do not they are considered revoked.

2. **Asynchronous sign-up quote verification** A validator can (at any time) ask IAS for attestation on a quote that another validator used to sign-up to check if his/her EPID key has been revoked since. If so the returned AVR will indicate that the key is revoked. A validator who obtains such an AVR from IAS can broadcast it in a blacklisting transaction, so that all the validators can check the veracity of the AVR and proceed with the blacklisting. To limit the use of blacklisting transactions as a means to thwart liveness for malicious validators one can control the rate at which they can be committed in different ways:

   - A certain number of participation tokens needs to be burned to commit a blacklisting transaction.
   - A validator can commit a blacklisting transaction only once he/she wins one or more elections.
   - A validator who commits a certain number of non-legit blacklisting transactions is blacklisted.

Security Considerations

1. T_{WT} **motivation**: A validator has at most T_{WT} seconds to consume a WaitTimer, namely obtain a WaitCertificate on it after the WaitTimer itself has expired. This constraint is enforced to avoid that in case there are no transactions to build a block for some time several validators might hold back after they waited the duration of their WaitTimers and generate the WaitCertificate only once enough transactions are available. At the point they will all send out their WaitCertificates generating a lot of traffic and possibly inducing forks. The timeout mitigates this problem.

2. **Enclave compromise:** a compromised SGX platform that is able to arbitrarily win elections cannot affect the correctness of the system, but can hinder progress by publishing void transactions. This problem is mitigated by limiting the frequency with which a validator (identified by his/her PPK) can win elections in a given time frame (see z-test documentation).

3. **WaitTimer duration manipulation:**

   (a) Imposing a c block participation delay after sign-up prevents validators from generating different pairs of OPK, PPK and pick the one that would result in the lowest value of the next WaitTimer duration as follows:

      i. Generate as many PPK,PSK pairs and therefore monotonic counters as possible.

      ii. Do not sign up but use all the enclaves (each using a different PPK, PSK and MCID) to create a WaitTimer every time a new block is committed until a very low duration is obtained (good chance of winning the election). Then collect all the different waitCertIds.

      iii. Ask each enclave to create the next waitTimer, whose duration depends on each of the different winning waitCertIds. Choose the PPK of the enclave giving me the lowest next duration and sign up with that.

      iv. As a result an attacker can win the first the election (with high probability) and can chain the above 3 steps to get a good chance of winning several elections in a row.

   (b) The nonce field in WaitCertificate is set to a random value so that a validator does not have control over the resulting $WaitCertId_n$. A validator winning an election could otherwise try different blockDigest input values to createWaitCertificate and broadcast the WaitCertificate whose $WaitCertId_n$ results in the lowest duration of his/her next WaitTimer.

   (c) The call `createWaitTimer()` in step 1 of the election phase (client side) is bound to the subsequent call to `createWaitCertificate()` by the internal state of the PoET enclave. More precisely only one call to `createWaitCertificate()` is allowed after a call to `createWaitTimer()` (and the duration has elapsed) as the value of the global active WaitTimer object activeWT is set to null at the end of `createWaitCertificate()` so that subsequent calls would fail. Therefore only one transaction block (identified by the input parameter blockDigest) can be attached to a WaitCertificate object. This prevents a malicious user from creating multiple WaitCertificates (each with a different nonce) resulting in different WaitCertId digests without re-creating a WaitTimer (and waiting for its duration) each time. It follows that as long as the duration of WaitTimer is not too small a malicious validator who wins the current election has very limited control over the duration of his/hers next WaitTimer.

(d) The check on the Monotonic Counter value guarantees only one enclave instance can obtain a WaitCertificate after the WaitTimer duration elapses. This again prevents a malicious user from running multiple instances of the enclave to create multiple WaitCertificates (each with a different nonce) resulting in different WaitCertId digests and selecting the one that would result in the lowest duration for a new WaitTimer.

(e) A monotonic counter with id MCID is created at the same time PPK and PSK are generated and the triple (MCID, PPK, PSK) is encrypted using AES-GCM with the Seal Key and saved in permanent storage. A malicious validator cannot run multiple enclave instances (before signing up) to create multiple monotonic counters without being forced to commit to using only one eventually. As a monotonic counter is bound to PPK, PSK through the AES-GCM encryption with the Seal Key, when a validator signs-up with a PPK it automatically commits to using the monotonic counter that was created along with PPK, PSK.

4. **Sign-up AEP replay:** the use of the nonce field in the AEP, which is set equal to $WaitCertId_n$, is used to prevent the replay of old AEPs.

Comments on Multi-user or Multi-ledger SGX Enclave Service

It is possible to use the same enclave for multiple users or ledgers by providing username and ledgername as input parameters to `generateSignUpData()` and `unsealSignUpData()`. Then the sign-up tuple (username, ledgername, PPK, PSK, MCID) is sealed to disk, with username and ledgername used to generate the filename. Anytime a user authenticates to the service the latter can have the enclave unseal and use the sign-up tuple from the file corresponding to that user (and ledger).

2.6.4 Population Size and Local Mean Computation

Parameters:

1. targetWaitTime: the desired average wait time. This depends on the network diameter and is selected to minimize the probability of a collision.
2. initialWaitTime: the initial wait time used in the bootstrapping phase until the ledger contains sampleLength blocks.
3. sampleLength: number of blocks that need to be on the ledger to finish the bootstrapping phase
4. minimumWaitTime: a lower bound on the wait time.

The population size is computed as follows:

```
1. $sumMeans = 0$
2. $sumWaits = 0$
3. **foreach** wait certificate $wc$ stored on the ledger:
       $sumWaits += wc.\text{duration} - \text{minimumWaitTime}$
       $sumMeans += wc.\text{localMean}$
4. $populationSize = sumMeans/sumWaits$
```

Assuming b is the number of blocks currently claimed, the local mean is computed as follows:

```
1. if $b <$ sampleLength then
```

$ratio = 1.0 \cdot b/\text{sampleLength}$ and
$\text{localMean} = \text{targetWaitTime} \cdot (1 - ratio^2) + \text{initialWaitTime} \cdot ratio^2$.

2. else $\text{localMean} = \text{targetWaitTime} \cdot \text{populationSize}$

2.6.5 z-test

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

Parameters:

1. zmax: test value, it measures the maximum deviation from the expected mean. It is selected so that the desired confidence interval $alpha$ is obtained. Example configurations are:
   (a) $\text{ztest} = 1.645 \leftrightarrow \alpha = 0.05$
   (b) $\text{ztest} = 2.325 \rightarrow \alpha = 0.01$
   (c) $\text{ztest} = 2.575 \rightarrow \alpha = 0.005$
   (d) $\text{ztest} = 3.075 \rightarrow \alpha = 0.001$
2. testValidatorId: the validator identifier under test.
3. blockArray: an array containing pairs of validator identifier and estimated population size. Each pair represents one published transaction block.
4. minObservedWins: minimum number of election wins that needs to be observed for the identifier under test.

The z-test is computed as follows:

```
observedWins = expectedWins = blockCount = 0
foreach block = (validatorId, populationEstimate) in blockArray:
    blockCount += 1
    expectedWins += 1 / populationEstimate

    if validatorId is equal to testValidatorId:
        observedWins += 1
        if observedWins > minObservedWins and observedWins > expectedWins:
            p = expectedWins / blockCount
            σ = sqrt(blockCount * p * (1.0 - p))
            z = (observedWins - expectedWins) / σ
            if z > zmax:
                return False
return True
```

If the z-test fails (False is returned) then the validator under test won elections at a higher average rate than expected.

2.7 Validator Network

The network layer is responsible for communication between validators in a Sawtooth network, including performing initial connectivity, peer discovery, and message handling. Upon startup, validator instances begin listening on a specified interface and port for incoming connections. Upon connection and peering, validators exchange messages with each other based on the rules of a gossip or epidemic[1] protocol.

A primary design goal is to keep the network layer as self-contained as possible. For example, the network layer should not need knowledge of the payload of application messages, nor should it need application-layer provided data

[1] http://web.mit.edu/devavrat/www/GossipBook.pdf

to connect to peers or to build out the connectivity of the network. Conversely, the application should not need to understand implementation details of the network in order to send and receive messages.

2.7.1 Services

The choice of 0MQ provides considerable flexibility in both available connectivity patterns and the underlying capabilities of the transport layer (IPv4, IPv6, etc.)

We have adopted the 0MQ Asynchronous Client/Server Pattern[2] which consists of a 0MQ ROUTER socket on the server side which listens on a provided endpoint, with a number of connected 0MQ DEALER sockets as the connected clients. The 0MQ guide describes the features of this pattern as follows:

- Clients connect to the server and send requests.
- For each request, the server sends 0 or more replies.
- Clients can send multiple requests without waiting for a reply.
- Servers can send multiple replies without waiting for new requests.

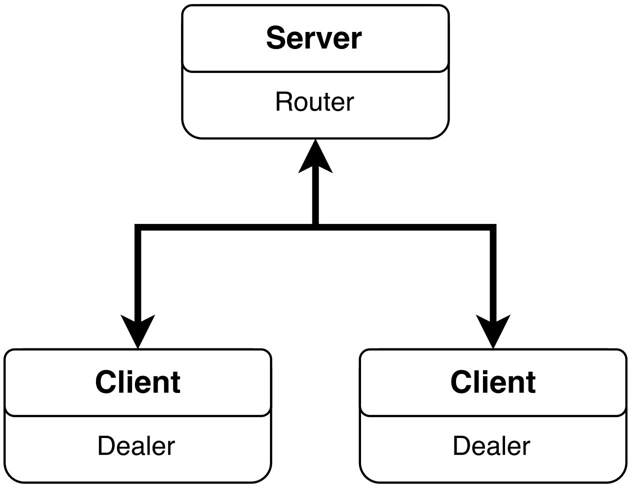


Fig. 1: Multiple DEALER to ROUTER socket pattern

2.7.2 States

We define three states related to the connection between any two validator nodes:

- Unconnected
- Connected - A connection is a required prerequisite for peering.
- Peered - A bidirectional relationship that forms the base case for application level message passing (gossip).

[2] http://zguide.zeromq.org/php:chapter3#toc19

2.7.3 Wire Protocol

We have standardized on protobuf serialization for any structured messages that need to be passed over the network. All payloads to or from the application layer are treated as opaque.

CONNECT

Connect is the mechanism for initiating the connection to the remote node. Connect performs a basic 0MQ DEALER->ROUTER connection to the remote node and exchanges identity information for the purpose of supporting a two-way conversation. Connections sit atop 0MQ sockets and allow the DEALER/ROUTER conversation.

PING

Ping messages allow for keep alive between ROUTER and DEALER sockets.

PEER

Peer requests establish a bidirectional peering relationship between the two nodes. A Peer request can be rejected by the remote node. If a peer request is rejected, the expectation is that a node attempts to connect with other nodes in the network via some strategy until the peering minimum connectivity threshold for that node is reached. If possible, the bi-directional relationship occurs over the already established 0MQ socket between DEALER and ROUTER.

GET_PEERS

Returns a list of peers of a given node. This can be performed in a basic Connected state and does not require peering to have occurred. The intent is to allow a node attempting to reach its minimum connectivity peering threshold to build a view of active candidate peers via a neighbor of neighbors approach.

BROADCAST(MSG)

Transmits an application message to the network following a ‘gossipy’ pattern. This does not guarantee 100% delivery of the message to the whole network, but based on the gossip parameters, nearly complete delivery is likely. A node only accepts messages for broadcast/forwarding from peers.

SEND(NODE, MSG)

Attempts to send a message to a particular node over the bidirectional 0MQ connection. Delivery is not guaranteed. If a node has reason to believe that delivery to the destination node is impossible, it can return an error response. A node only accepts a message for sending from peer nodes.

REQUEST(MSG)

A request is a special type of broadcast message that can be examined and replied to, rather than forwarded. The intent is for the application layer to construct a message payload which can be examined by a special request handler and replied to, rather than forwarded on to connected peers. If the application layer reports that the request can’t be satisfied, the message will be forwarded to peers per the rules of a standard broadcast message. A node only accepts request messages from peer nodes.

UNPEER

Breaks the peering relationship between nodes. This may occur in several scenarios, for example a node leaving the network (nodes may also silently leave the network, in which case their departure will be detected by the failure of the ping/keepalive). An unpeer request does not necessarily imply a disconnect.

DISCONNECT

Breaks the wire protocol connection to the remote node. Informs the ROUTER end to clean up the connection.

2.7.4 Peer Discovery

A bidirectional peering via a neighbor of neighbors approach gives reliable connectivity (messages delivered to all nodes >99% of the time based on random construction of the network).

Peer connections are established by collecting a suitable population of candidate peers through successive CONNECT/GET_PEERS calls (neighbors of neighbors). The connecting validator then selects a candidate peer randomly from the list and attempts to connect and peer with it. If this succeeds, and the connecting validator has reached minimum connectivity, the process halts. If minimum connectivity has not yet been reached, the validator continues attempting to connect to new candidate peers, refreshing its view of the neighbors of neighbors if it exhausts candidates.

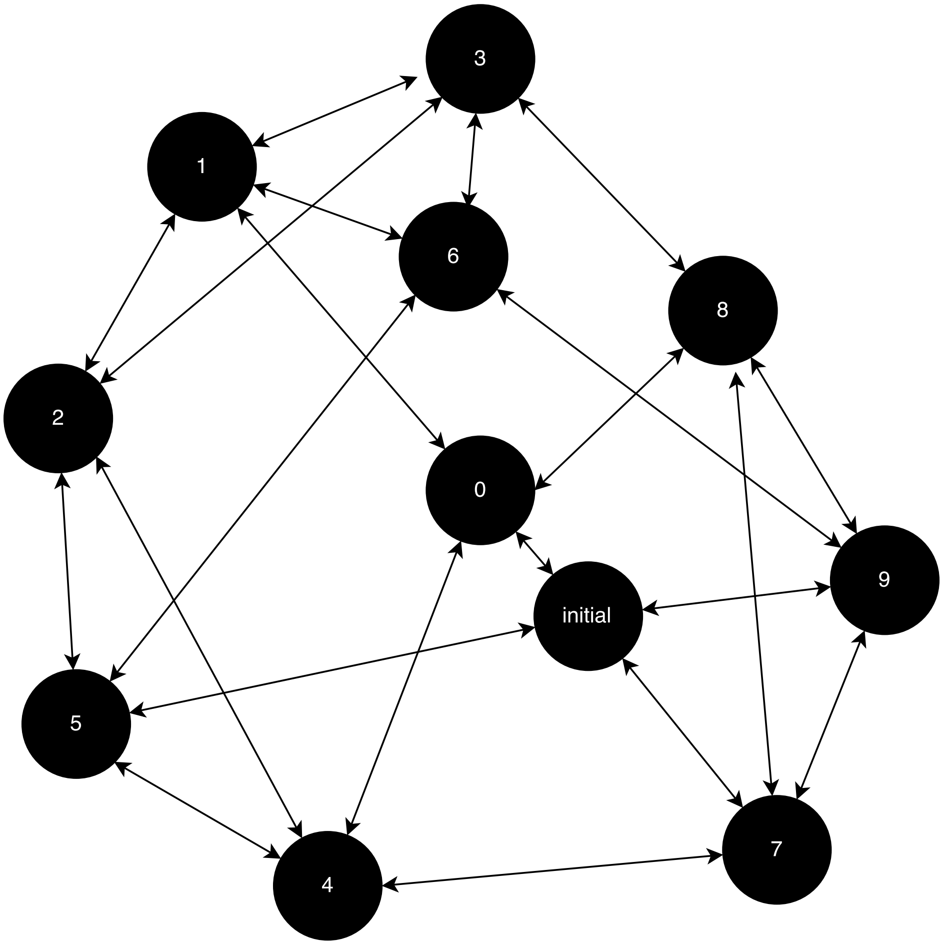
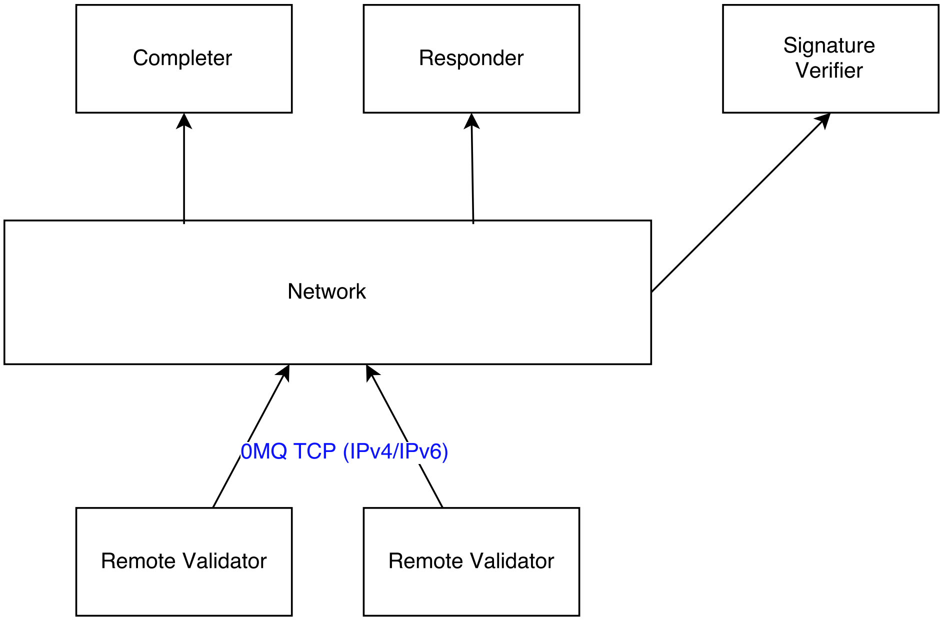


Fig. 2: Output of bidirectional peering with targeted connectivity of 4.

The network component continues to perform a peer search if its number of peers is less than the minimum connectivity. The network component rejects peering attempts if its number of peers is equal to or greater than the maximum connectivity. Even if maximum peer connections is reached, a network service should still accept and respond to a reasonable number of connections (for the purposes of other node topology build outs, etc.)

2.7.5 Related Components



2.7.6 Message Delivery

The network delivers application messages (payloads received via BROADCAST or SEND) to the application layer. The network also performs a basic validation of messages prior to forwarding by calling a handler in the Message Validation component.

When the network receives a REQUEST message, it calls a provided handler (a "Responder", for example) to determine if the request can be satisfied. If so, the expectation is that the application layer generates a SEND message with a response that satisfies the request. In this condition, the network layer does not continue to propagate the REQUEST message to the network.

In the case where a node could not satisfy the request, the node stores who it received the request from and BROADCASTs the request on to its peers. If that node receives a SEND message with the response to the request, it forwards the SEND message back to the original requester.

The network accepts application payloads for BROADCAST, SEND, and REQUEST from the application layer.

2.7.7 Network Layer Security

0MQ includes a TLS[3] like certificate exchange mechanism and protocol encryption capability which is transparent to the socket implementation. Support for socket level encryption is currently implemented with server keys being read from the validator.toml config file. For each client, ephemeral certificates are generated on connect. If the server key pair is not configured, network communications between validators will not be authenticated or encrypted.

2.7.8 Network Permissioning

One of the permissioning requirements is that the validator network be able to limit the nodes that are able to connect to it. The permissioning rules determine the roles a connection is able to play on the network. The roles control the types of messages that can be sent and received over a given connection. The components and nodes that wish to take on these roles must participate in an authorization "handshake" and request the roles they want to take on. The entities acting in the different roles will be referred to as requesters below.

[3] https://github.com/zeromq/pyzmq/blob/master/examples/security/ironhouse.py

Validators are able to determine whether messages delivered to them should be handled or dropped based on a set of role and identities stored within the Identity namespace. Each requester will be identified by the public key derived from their identity signing key. Permission verifiers examine incoming messages against the policy and the current configuration and either permit, drop, or respond with an error. In certain cases, the connection will be forcibly closed – for example: if a node is not allowed to connect to the validator network.

The following describes the procedure for establishing a new connection with the validator. The procedure supports implementing different authorization types that require the requester to prove their identity. If a requester deviates from the procedure in any way, the requester will be rejected and the connection will be closed. The same is true if the requester sends multiple ConnectionRequests or multiple of any authorization-type message. Certain low level messages, such as Ping, can be used before the procedure is complete, but are rate limited. If too many of the low level messages are received or they are received too close together, the connection may be considered malicious and rejected.

The validator receiving a new connection receives a ConnectionRequest. The validator responds with a ConnectionResponse message. The ConnectionResponse message contains a list of RoleEntry messages and an AuthorizationType. Role entries are the accepted type of connections that are supported on the endpoint that the ConnectionRequest was sent to. AuthorizationType describes the procedure required to gain access to that role. Trust is the simplest authorization type and must be implemented by all requesters at a minimum. If the requester cannot comply with the given authorization type for that role entry, it is unable to gain access to that role.

```
message ConnectionRequest {
  // This is the first message that must be sent to start off authorization.
  // The endpoint of the connection.
  string endpoint = 1;
}

enum RoleType {
  // A shorthand request for asking for all allowed roles.
  ALL = 0;

  // Role defining validator to validator communication
  NETWORK = 1;
}

message ConnectionResponse {
  // Whether the connection can participate in authorization
  enum Status {
    OK = 0;
    ERROR = 1;
  }

  //Authorization Type required for the authorization procedure
  enum AuthorizationType {
    TRUST = 0;
    CHALLENGE = 1;
  }

  message RoleEntry {
    // The role type for this role entry
    RoleType role = 1;

    // The Authorization Type required for the above role
    AuthorizationType auth_type = 2;
  }

  repeated RoleEntry roles = 1;
```

(continues on next page)

(continued from previous page)

```
  Status status = 2;
}
```

In the future, RoleType will include other roles such as CLIENT, STATE_DELTA, and TP.

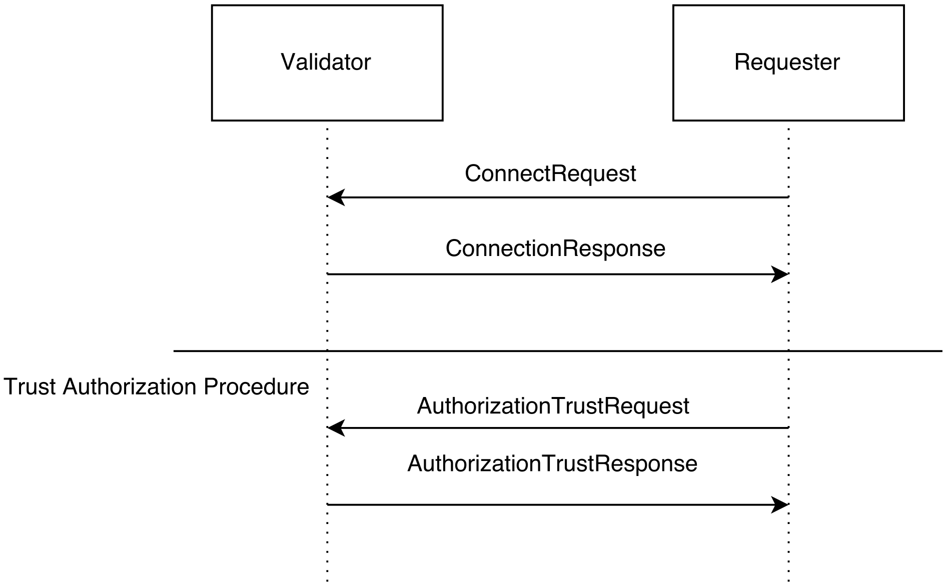
Authorization Types

Presented here are the two authorization types that will be implemented initially: Trust and Challenge.

Trust The simplest authorization type is Trust. If Trust authorization is enabled, the validator will trust the connection and approve any roles requested that are available on that endpoint. If the requester wishes to gain access to every role it has permission to access, it can request access to the role ALL, and the validator will respond with all available roles. However, if a role that is not available is requested, the requester is rejected and the connection will be closed.

```
message AuthorizationTrustRequest {
  // A set of requested RoleTypes
  repeated RoleType roles = 1;
  string public_key = 2;
}

message AuthorizationTrustResponse {
  // The actual set the requester has access to
  repeated RoleType roles = 1;
}
```

Message flow for Trust Authorization:


Validator
Requester
ConnectRequest
ConnectionResponse
Trust Authorization Procedure
AuthorizationTrustRequest
AuthorizationTrustResponse

Challenge If the connection wants to take on a role that requires a challenge to be signed, it will request the challenge by sending the following to the validator it wishes to connect to.

```
message AuthorizationChallengeRequest {
  // Empty message sent to request a payload to sign
}
```

The validator will send back a random payload that must be signed.

```
message AuthorizationChallengeResponse {
  // Random payload that the connecting node must sign
  bytes payload = 1;
}
```

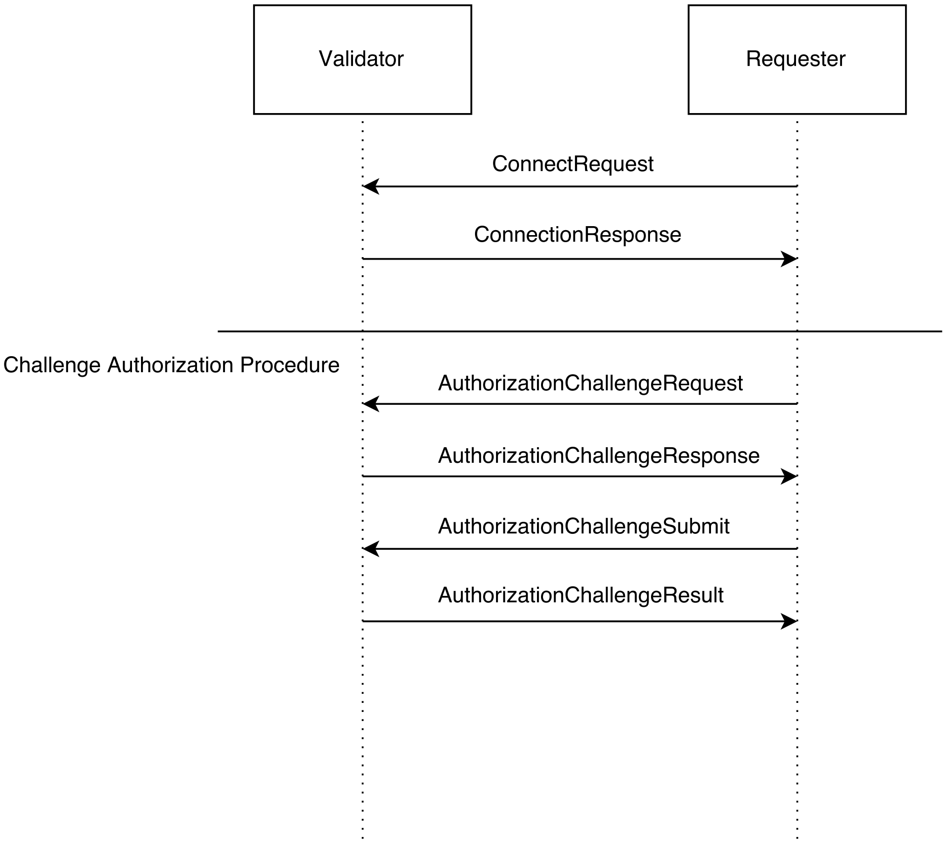
The requester then signs the payload message and returns a response that includes the following:

```
message AuthorizationChallengeSubmit {
  // public key of node
  string public_key = 1;

  // signature derived from signing the challenge payload
  string signature = 3;

  // A set of requested Roles
  repeated RoleType roles = 4;
}
```

The requester may also request ALL. The validator will respond with a status that says whether the challenge was accepted and the roles that the connection is allowed take on.

```
message AuthorizationChallengeResult {
  // The approved roles for that connection
  repeated RoleType roles = 1;
}
```

Message flow for Challenge Authorization:


Validator
Requester
ConnectRequest
ConnectionResponse
Challenge Authorization Procedure
AuthorizationChallengeRequest
AuthorizationChallengeResponse
AuthorizationChallengeSubmit
AuthorizationChallengeResult

When the validator receives an AuthorizationChallengeSubmit message, it verifies the public key against the signature. If the public key is verified, the requested roles is checked against the stored roles to see if the public key is included in the policy. If the node's response is accepted, the node's public key is stored and the requester may start sending messages for the approved roles.

If the requester wanted a role that is either not available on the endpoint or the requester does not have access to one of the roles requested, the challenge will be rejected and the connection is closed. At that point the requester will need to restart the connection process.

Authorization Violation If at any time a requester tries to send a message that is against its allowed permission, the validator responds with an AuthorizationViolation message and the connection is closed. If that requester wishes to rejoin the network, it will need to go back through the connection and authorization process described above.

```
message AuthorizationViolation {
  // The Role the requester did not have access to
  RoleType violation = 1;
}
```

2.8 Permissioning Requirements

The following is a detailed list of permissioning requirements, capabilities, and user stories that aid in the explanation and discussion of the permissioning design for Hyperledger Sawtooth.

The following permission groups are presented in this document:

- Transactor key permissioning, which controls acceptance of transactions and batches based on signing keys.
- Validator key permissioning, which controls which nodes are allowed to establish connections to the validator network.

The permissioning types are further broken down into capability requirements. Each requirement is made up of a short description, the network scenario the requirement supports, and associated user stories. User stories are short statements about a requirement or capability that an actor desires. These short descriptions illustrate the need for specific permission requirements.

The design requirements for permissioning include both local validator configuration and network-wide on-chain permissioning. Local configuration is needed so that a validator can limit who is allowed to submit batches and transactions directly to that validator. This is useful, for example, if a company is running its own validator and wishes to only allow members of that company to submit transactions. On-chain configuration is required so that the whole of the network can enforce consistent permissioning rules. For example, in validator key permissioning, all the validators on the network will reject the same node from peering, ensuring that only known nodes can join the network.

2.8.1 Definitions

General Definitions

The following list contains architecture-level definitions for Sawtooth concepts. For general definitions, see the *Glossary*.

Authorization procedure "Handshake" that a connection must go through to be allowed to connect to a validator. If the connection does not follow the authorization procedure correctly, the connection will be rejected and closed.

Batch Protobuf object containing a list of transactions, a header, and a signature. The signature is generated by the batch signer by signing the batch header with the batch signer's private key. For more information on the structure of a batch, see *Batch Data Structure*.

Client Program that connects to a validator and is able to query the state of the blockchain and submit batches.

Policy Set of DENY and PERMIT rules that can be used to permission the access to the validator network and which transactors can participate on the network.

Transaction Protobuf object containing a payload, header, and signature. The signature is generated by the transaction signer by signing the transaction header with the transaction signer's private key. For more information on the structure of a transaction, see *Transaction Data Structure*.

Transaction Family Set containing a transaction payload format, a model for storing information in global state, and a procedure for validating a transaction and updating state based on the transaction payload. For information on specific transaction families, see *Transaction Family Specifications*.

Actor Definitions

The following terms are used for the actors within a Sawtooth network:

Transactor Person or program which signs batches or transactions. When signing a batch, the transactor is also a batch signer. When signing a transaction, a transactor is also a transaction signer.

Batch signer Transactor that signs a batch header using the batch signer's private key. The resulting signature is included as part of the batch. The batch signer's public key is also stored in the batch header.

Transaction Signer Transactor that signs a transaction header using the transaction signer's private key. The resulting signature is included as part of the transaction. The transaction signer's public key is also stored in the transaction header.

Network Operator One or more people who collectively manage the Sawtooth network. The network operator determines and coordinates the overall network characteristics, such as determining which transaction processors are being run, setting the consensus mechanism, modifying on-chain settings, and modifying on-chain roles.

Sysadmin System administrator; a person who installs and configures the validator software and configures the validator software using config files. The role of sysadmin does not include activities that modify on-chain settings.

2.8.2 Validator Network Scenarios

The following example network scenarios are used to illustrate when each capability requirement would be useful.

Public Network

In a public network, all connections will be allowed and all transactors are allowed to sign batches and transactions.

In order for a public Sawtooth network to function correctly, an incentive system would also need to be designed and implemented. Such a system is beyond the scope of this design and so it is omitted.

Consortium Network

In a consortium network, only specific validators will be allowed to join the validator network and participate in consensus. The network will still allow any client, transaction processor, or state delta subscriber to connect to a validator and accept batches and transactions signed by all transactors.

Private Network

In a private network, only specific validators will be allowed to join the validator network and participate in consensus. The validators in the network will only accept connections from specific clients and will control if the client is allowed to submit batches and query specific address prefixes in state. Only specific transaction processors and state delta subscribers will be allowed to connect to the local validator. Batches and transactions received by the validator can only be signed by permitted transactors. Transactors may also be restricted to only sending transactions for certain transaction families.

Network Scenario	Capabilities
Public Network	• Allow all batch signers to submit batches • Allow all transaction signers to submit transactions • Allow all nodes to join the validator network
Consortium Network	• Allow all batch signers to submit batches • Allow all transaction signers to submit transactions • Allow only specific nodes to join the validator network • Allow only specific nodes to participate in consensus • Support policy-based transactor permissioning
Private Network	• Allow only specific batch signers to submit batches • Allow only specific transaction signers to submit transactions • Allow only specific nodes to join the validator network • Allow only specific nodes to participate in consensus • Restrict the type of transactions transactors can sign • Restrict address space access to a limited set of transactors • Support policy-based transactor permissioning

2.8.3 Transactor Key Permissioning

The following stories are related to permissioning performed on the basis of a transactor's public signing key. This includes both batch signers and transaction signers. The validator will check local configuration and network configuration when receiving a batch from a client and only batch signers permitted in the intersection of the two configurations will be allowed. When the validator receives a batch from a peer, only the network configuration will be checked. This is required to prevent network forks.

Allow all batch signers to submit batches

Network Scenario	• Public - YES • Consortium - YES • Private - NO
Description	The validator must be able to be configured to allow all batches to be submitted, regardless of who the batch signer is. In other words, if a client is connected to the validator and that client is allowed to submit batches, the batches will not be rejected due to the public key that was used to sign the batch. These batches will still be rejected if they fail signature verification.
User Stories	• A sysadmin must be able to configure a local validator to accept batches signed by any batch signer. • A network operator must be able to configure the validator network to accept batches signed by any batch signer.

Allow only specific batch signers to submit batches

Network Scenario	• Public - NO • Consortium - NO • Private - YES
Description	The validator must be able to be configured to only allow certain batch signers to submit batches. If the validator receives a batch that was signed by a batch signer whose public key is not allowed, that batch should be dropped. Batches should also be checked before block validation. If the validator network permits a given batch signer, the validator must accept batches signed by that batch signer from peers, regardless of its local configuration.
User Stories	• A sysadmin must be able to configure a local validator to accept batches signed only by predefined batch signers. • A network operator must be able to configure the whole validator network to only accept batches from specific batch signers.

Allow all transaction signers to submit transactions

Network Scenario	• Public - YES • Consortium - YES • Private - NO
Description	The validator must be able to be configured to allow all transactions to be submitted, regardless of who the transaction signer is. In other words, if a client is connected to the validator and the client is allowed to submit transactions, the transactions will not be rejected due to the public key that was used to sign the transactions. These transactions will still be rejected if they fail signature verification.
User Stories	• A sysadmin must be able to configure a local validator to accept transactions signed by any transaction signer. • A network operator must be able to configure the whole validator network to accept transactions signed by any batch signer.

Allow only specific transaction signers to submit transactions

Network Scenario	• Public - NO • Consortium - NO • Private - YES
Description	The validator must be able to be configured to only allow certain transaction signers to submit transactions. In other words, if the validator receives a transaction that was signed by a transaction signer whose public key is not allowed, that transaction should be dropped. Transactions should also be checked during block validation. If the validator network permits a given transaction signer, the validator must accept transaction signed by that transaction signer from peers, regardless of its local configuration.
User Stories	• A sysadmin must be able to configure a local validator to accept transactions signed only by predefined transaction signers. • A network operator must be able to configure the whole validator network to accept batches only from specific transaction signers.

Restrict the type of transactions transactors can sign

Network Scenario	• Public - NO • Consortium - NO • Private - YES
Description	The validator must be able to restrict the transaction signers that are allowed to sign transactions for a specific transaction family. This permissioning would be separate from any public key permissioning enforced by the transaction family logic.
User Stories	• A network operator must be able to configure the whole validator network to only accept transactions that were signed by allowed transaction signers for a specific transaction family.

Support policy-based transactor permissioning

Network Scenario	• Public - NO • Consortium - YES • Private - YES
Description	The validator must be able to enforce transactor permissions based on defined policies.
User Stories	• A sysadmin must be able to configure a local validator to only accept transactions and batches that are signed by transactors that are allowed by predefined locally stored policies. • A network operator must be able to configure the whole validator network to only accept transactions and batches that are signed by transactor that are allowed by defined policies.

2.8.4 Validator Key Permissioning

The following stories are related to permissioning performed on the basis of a node's public signing key.

Allow all nodes to join the validator network

Network Scenario	• Public - YES • Consortium - NO • Private - NO
Description	The validator network must be able to be configured to allow all validator nodes to join the network. This means that every validator on the network should accept the node as a peer regardless of what its public key is. The validator nodes will also be able to participate in consensus and communicate over the gossip protocol. The node will still need to go through the authorization procedure defined by the validator the node is trying to connect to. If, for any reason, the node fails the authorization procedure, it will be rejected.
User Stories	• A network operator must be able to configure the validator network to accept all nodes that wish to connect, regardless of the node's public key.

Allow only specific nodes to join the validator network

Network Scenario	• Public - NO • Consortium - YES • Private - YES
Description	The validator network must be able to be configured to only allow nodes to join the network if the node's public key is permitted. In other words, if a validator receives a connection request from a node, and the validator does not recognize the node's public key, the connection should be rejected.
User Stories	• A network operator must be able to configure the validator network to only accept connections from nodes with specific public keys.

Allow only specific nodes to participate in consensus

Network Scenario	• Public - NO • Consortium - YES • Private - YES
Description	The validator network must be able to be configured to only allow specific nodes to participate in consensus. This is separate from any restrictions enforced by the consensus algorithm. The nodes that are not allowed to participate in consensus can still validate blocks but are not allowed to publish blocks.
User Stories	• A network operator must be able to configure the validator network to only allow certain nodes to participate in consensus.

2.9 Injecting Batches and On-Chain Block Validation Rules

Injecting transactions into blocks by the validator is required to support a variety of use cases such as:

- Setting information in state for use by transaction processors that cannot reasonably be provided with every transaction. For example, setting the block number or timestamp.
- Automatically submitting transactions in response to a particular state. For example, submitting bond quote matching transactions.
- Testing automation. For example, programming the compute time of test transactions to increase after a certain number of batches.

Imposing a set of rules on the relationships between transactions, or inter-transaction validation, is also required for some use cases. Two examples of inter-transaction validation would be:

- Only N of transaction type X may be included in a block.
- Transaction type X may only occur at position Y in a block.

2.9.1 BatchInjector Interface

The BatchInjector class supports injecting transactions into blocks:

```
interface BatchInjector:
  // Called when a new block is created and before any batches are added. A list of
  // batches to insert at the beginning of the block must be returned. A StateView
  // is provided for inspecting state as of the previous block.
  block_start(string previous_block_id) -> list<Batch>

  // Called before inserting the incoming batch into the pending queue for the
  // given block. A list of batches to insert before this batch must be returned.
  before_batch(string previous_block_id, Batch batch) -> list<Batch>

  // Called after inserting the incoming batch into the pending queue for the
  // given block. A list of batches to insert after this batch must be returned.
  after_batch(string previous_block_id, Batch batch) -> list<Batch>
```

(continues on next page)

(continued from previous page)

```
// Called just before finalizing and completing a Block. An ordered list of batches
// that will be committed in the block is passed in. A list of batches to insert at
// the end of the block must be returned.
block_end(string previous_block_id, list<Batch> batches) -> list<Batch>
```

The Journal will call each of the methods at the appropriate times for all included BatchInjectors, thus injecting batches into the new block.

2.9.2 On-Chain Configuration

The set of BatchInjectors to load is configured with an on-chain setting; this is similar to configuring the consensus module loaded by the validator. The sawtooth.validator.batch_injectors setting key stores a comma-separated list of batch injectors to load. This list is parsed by the validator at the beginning of block publishing for each block and the appropriate injectors are loaded.

This setting is controlled using the existing Settings Transaction Family. Take care when when updating this setting, because an incorrect value may cause transaction families to behave incorrectly.

2.9.3 On-Chain Validation Rules

An on-chain setting holds a set of validation rules that are enforced for each block. On-chain validation rules will be stored as a string in the setting key sawtooth.validator.block_validation_rules. Rules will be enforced by the block validator.

A simple syntax will be used for specifying validations rules as defined by the following.

- A validation rule consists of a name followed by a colon and a comma-separated list of arguments: `rulename:arg,arg,...,arg`
- Separate multiple rules with semicolons: `rulename1:arg,arg,...,arg;rulename2:arg,arg,...,arg`
- Spaces, tabs, and newlines are ignored.

The following rules are defined:

NofX Only N transaction of transaction type X may be included in a block. The first argument must be an integer. The second argument is the name of a transaction family. For example, the string `NofX:2,intkey` means allow only two IntegerKey transactions per block.

XatY A transaction of type X must be in the block at position Y. The first argument is interpreted as the name of a transaction family. The second argument must be interpretable as an integer and defines the index of the transaction in the block that must be checked. Negative numbers can be used and count backwards from the last transaction in the block. The first transaction in the block has index 0. The last transaction in the block has index -1. If abs(Y) is larger than the number of transactions per block, then there would not be a transaction of type X at Y and the block would be invalid. For example, the string `XatY:intkey,0` means the first transaction in the block must be an IntegerKey transaction.

local A transaction must be signed by the same key as the block. This rule takes a list of transaction indices in the block and enforces the rule on each. This rule is useful in combination with the other rules to ensure a client is not submitting transactions that should only be injected by the winning validator.

2.9.4 Example: BlockInfoInjector

The BlockInfoInjector inserts a BlockInfo transaction at the beginning of every block. The transaction updates state with information about the block that was just committed as well as a timestamp. For more information, see the *BlockInfo Transaction Family*

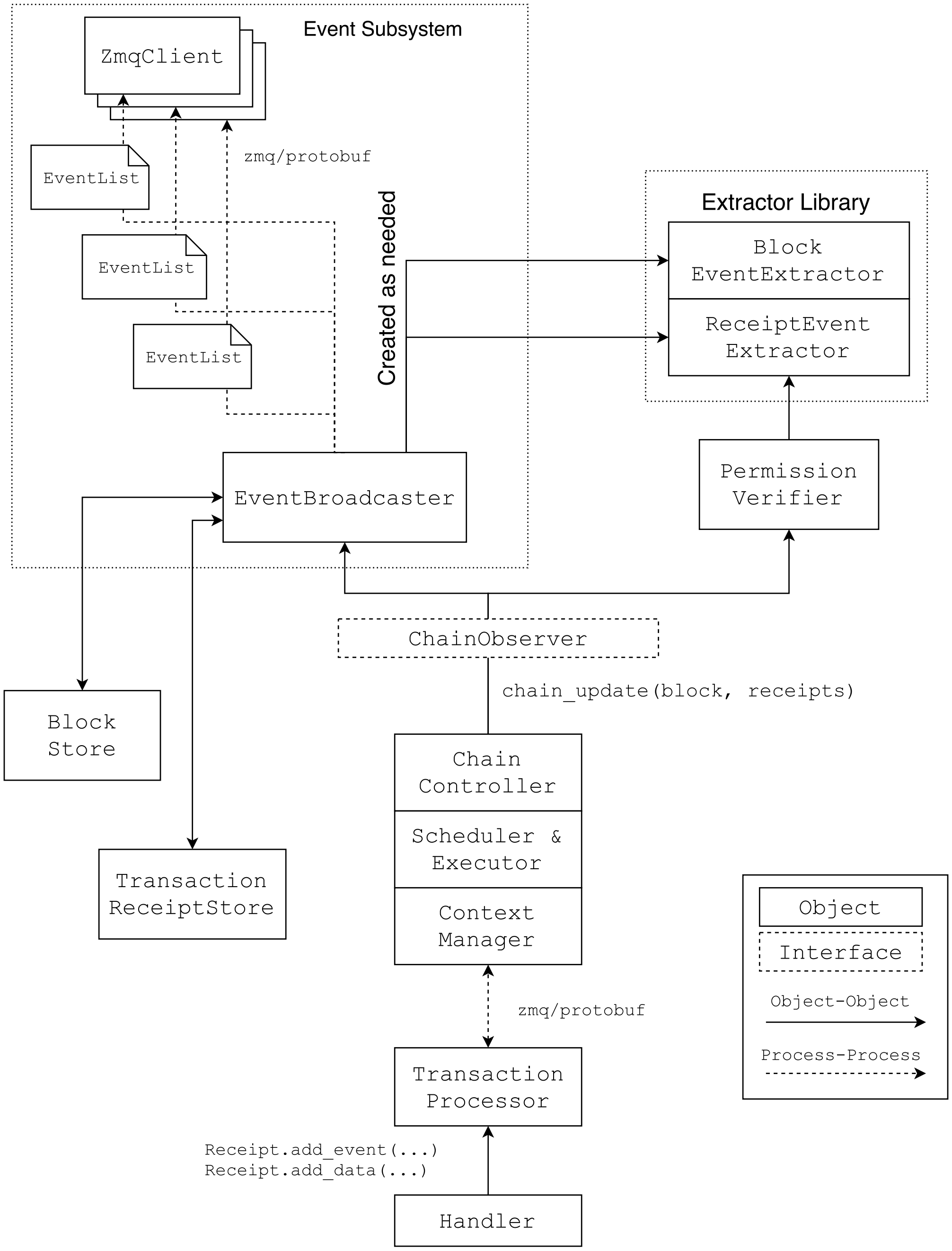
The following validation rules are added to the set of on-chain validation rules in order to prevent bad actors from injecting incorrect but valid BlockInfo transactions. The rules require that only one BlockInfo transaction is included per block, that the transaction is at the beginning of the block, and that the transaction is signed by the same key that signed the block.

- NofX:1,block_info;
- XatY:block_info,0;
- local:0

2.10 Events and Transaction Receipts

Hyperledger Sawtooth supports creating and broadcasting events. This allows applications to do the following:

- Subscribe to events that occur related to the blockchain, such as a new block being committed or switching to a new fork.
- Subscribe to application specific events defined by a transaction family.
- Relay information about the execution of a transaction back to clients without storing that data in state.



2.10.1 Events

Events are represented with the following protobuf message:

```
message Event {
  // Used to subscribe to events and servers as a hint for how to deserialize
  // event_data and what pairs to expect in attributes.
  string event_type = 1;

  // Transparent data defined by the event_type.
  message Attribute {
```

(continues on next page)

(continued from previous page)

```
    string key = 1;
    string value = 2;
  }
  repeated Attribute attributes = 2;

  // Opaque data defined by the event_type.
  bytes  data = 3;
}
```

Events are extracted from other data structures such as blocks or transaction receipts. In order to treat this extraction uniformly, an EventExtractor interface is implemented for each event source. The EventExtractor interface takes a list of EventSubscriptions and will only generate events that are in the union of all subscriptions. An event is "in a subscription" if the event's event_type field matches the subscription's event_type field and the filter's key-value pair (if the subscription has any filters) matches a key-value pair in the event's attribute field.

```
interface EventExtractor:
  // Construct all the events of interest by taking the union of all subscriptions.
  // One extractor should be created for each input source that events can be
  // extracted from. This input source should be passed to the implementation through
  // the constructor.
  extract(list<EventSubscription> subscriptions) -> list<Event>

// If no subscriptions of a given event_type are passed to EventExtractor.extract,
// the extractor does not need to return events of that type.
class EventSubscription:
  string event_type
  list<EventFilter> filters
```

The following filters are implemented:

SIMPLE_ANY Represents a subset of events within an event type.

Since multiple event attributes with the same key can be present in an event, an event is considered part of this filter if its match string matches the value of ANY attribute with the filter's key.

For example, if an event has the following attributes:

- Attribute(key="address", value="abc")
- Attribute(key="address", value="def")

it will pass the following filter:

SimpleAnyFilter(key="address", match_string"abc")

Because it matches one of the two attributes with the key "address".

SIMPLE_ALL Represents a subset of events within an event type.

Since multiple event attributes with the same key can be present in an event, an event is considered part of this filter if its match string matches the value of ALL attribute with the filter's key.

For example, if an event has the following attributes:

- Attribute(key="address", value="abc")
- Attribute(key="address", value="def")

it will NOT pass this filter:

SimpleAllFilter(key="address", value="abc")

Because it does not match all attributes with the key "address".

REGEX_ANY Represents a subset of events within an event type. Pattern must be a valid regular expression that can be compiled by the re module.

Since multiple event attributes with the same key can be present in an event, an event is considered part of this filter if its pattern matches the value of ANY attribute with the filter's key.

For example, if an event has the following attributes:

- Attribute(key="address", value="abc")
- Attribute(key="address", value="def")

it will pass the following filter:

> AnyRegexFilter(key="address", value="abc")

Because it matches one of the two attributes with the key "address".

REGEX_ALL Represents a subset of events within an event type. Pattern must be a valid regular expression that can be compiled by the re module.

Since multiple event attributes with the same key can be present in an event, an event is considered part of this filter if its pattern matches the value of ALL attribute with the filter's key.

For example, if an event has the following attributes:

- Attribute(key="address", value="abc")
- Attribute(key="address", value="def")

it will NOT pass this filter:

> AllRegexFilter(key="address", value="abc")

Because it does not match all attributes with the key "address".

An EventBroadcaster manages external event subscriptions and forwards events to subscribers as they occur. In order for the EventBroadcaster to learn about events, the ChainController class implements the Observer pattern. The ChainController acts as the subject and observers of the ChainController implement the ChainObserver interface.

```
interface ChainObserver:
  // This method is called by the ChainController on block boundaries.
  chain_update(Block block, list<TransactionReceipt> receipts)

class EventBroadcaster:
  // Register the subscriber for the given event subscriptions and begin sending it
  // events on block boundaries.
  //
  // If any of the block ids in last_known_block_ids are part of the current chain,
  // the observer will be notified of all events that it would have received based on
  // its subscriptions for each block in the chain since the most recent
  // block in last_known_block_ids.
  //
  // Raises an exception if:
  // 1. The subscription is unsuccessful.
  // 2. None of the block ids in last_known_block_ids are part of the current chain.
  add_subscriber(string connection_id, list<EventSubscription> subscriptions,
                 list<string> last_known_block_ids)

  // Stop sending events to the subscriber
  remove_subscriber(string connection_id)

  // Notify all observers of all events they are subscribed to.
  chain_update(Block block, list<TransactionReceipt> receipts)
```

On receiving a chain_update() notification from the ChainController, the EventBroadcaster instantiates a new EventExtractor, passes each extractor all the EventSubscriptions for all subscribers, and receives the list of events that is the union of the events that all subscribers are interested in. The EventBroadcaster then distributes the events to each subscriber based on the subscriber's list of subscriptions.

To reduce the number of messages sent to subscribers, multiple Event messages are wrapped in an EventList message when possible:

```
EventList {
  repeated Event events = 1;
}
```

ClientEventSubscribeRequest messages are sent by external clients to the validator in order to subscribe to events. ClientEventSubscribeResponse messages are sent by the validator to the client in response to notify the client whether their subscription was successful. When an external client subscribes to events, they may optionally send a list of block ids along with their subscriptions. If any of the blocks sent are in the current chain, the EventBroadcaster will bring the client up to date by sending it events for all blocks since the most recent block sent with the subscribe request.

```
message ClientEventsSubscribeRequest {
    repeated EventSubscription subscriptions = 1;
    // The block id (or ids, if trying to walk back a fork) the subscriber last
    // received events on. It can be set to empty if it has not yet received the
    // genesis block.
    repeated string last_known_block_ids = 2;
}

message ClientEventsSubscribeResponse {
    enum Status {
         OK = 0;
         INVALID_FILTER = 1;
         UNKNOWN_BLOCK = 2;
    }
    Status status = 1;
    // Additional information about the response status
    string response_message = 2;
}
```

Event Extractors

Two event extractors are created to extract events from blocks and transaction receipts: BlockEventExtractor and ReceiptEventExtractor. The BlockEventExtractor will extract events of type “sawtooth/block-commit”. The ReceiptEventExtractor will extract events of type “sawtooth/state-delta” and events defined by transaction families.

Example events generated by BlockEventExtractor:

```
// Example sawtooth/block-commit event
Event {
  event_type = "sawtooth/block-commit",
  attributes = [
    Attribute { key = "block_id", value = "abc...123" },
    Attribute { key = "block_num", value = "523" },
    Attribute { key = "state_root_hash", value = "def...456" },
    Attribute { key = "previous_block_id", value = "acf...146" },
  ],
}
```

Example events generated by ReceiptEventExtractor:

```
// Example transaction family specific event
Event {
  event_type = "xo/create",
  attributes = [Attribute { key = "game_name", value = "game0" }],
}

// Example sawtooth/block-commit event
Event {
  event_type = "sawtooth/state-delta",
  attributes = [Attribute { key = "address", value = "abc...def" }],
  event_data = <bytes>
}
```

2.10.2 Transaction Receipts

Transaction receipts provide clients with information that is related to the execution of a transaction but should not be stored in state, such as:

- Whether the transaction was valid
- How the transaction changed state
- Events of interest that occurred during execution of the transaction
- Other transaction-family-specific execution information

Transaction receipts can also provide the validator with information about transaction execution without re-executing the transaction.

Sawtooth transaction receipts are represented as a protobuf message when exposed to clients. A transaction receipt contains a list of StateChange messages, a list of Event messages, and a list of TransactionReceipt.Data messages:

```
message TransactionReceipt {
  // State changes made by this transaction
  // StateChange is already defined in protos/state_delta.proto
  repeated StateChange state_changes = 1;
  // Events fired by this transaction
  repeated Event events = 2;
  // Transaction family defined data
  repeated bytes data = 3;

  string transaction_id = 4;
}
```

The fields in the TransactionReceipt.Data are opaque to Sawtooth and their interpretation is left up to the transaction family. TransactionReceipt.Data can be requested for a transaction that has been committed and should only be used to store information about a transaction's execution that should not be kept in state. Clients can use the event system described above to subscribe to events produced by transaction processors.

Transaction Receipt Store

The TransactionReceiptStore class stores receipts. New receipts are written to the TransactionReceiptStore after a block is validated.

Transaction receipts will only be stored in this off-chain store and will not be included in the block. Note that because a transaction may exist in multiple blocks at a time, the transaction receipt is stored by both transaction id and block state root hash.

```
class TransactionReceiptStore:
      def put_receipt(self, txn_id, receipt):
      """Add the given transaction receipt to the store. Does not guarantee
              it has been written to the backing store.

      Args:
              txn_id (str): the id of the transaction being stored.
              state_root_hash: the state root of the block this transaction was
          executed in.
              receipt (TransactionReceipt): the receipt object to store.
      """

      def get_receipt(self, txn_id):
      """Returns the TransactionReceipt

      Args:
              txn_id (str): the id of the transaction for which the receipt
          should be retrieved.
              state_root_hash: the state root of the block this transaction was
          executed in.

      Returns:
              TransactionReceipt: The receipt for the given transaction id.

      Raises:
              KeyError: if the transaction id is unknown.
      """
```

Message Handlers

Once transaction receipts are stored in the TransactionReceiptStore, clients can request a transaction receipt for a given transaction id.

```
// Fetches a specific txn by its id (header_signature) from the blockchain.
message ClientReceiptGetRequest {
    repeated string transaction_ids = 1;
}

// A response that returns the txn receipt specified by a
// ClientReceiptGetRequest.
//
// Statuses:
//   * OK - everything worked as expected, txn receipt has been fetched
//   * INTERNAL_ERROR - general error, such as protobuf failing to deserialize
//   * NO_RESOURCE - no receipt exists for the transaction id specified
message ClientReceiptGetResponse {
    enum Status {
        OK = 0;
        INTERNAL_ERROR = 1;
        NO_RESOURCE = 4;
    }
    Status status = 1;
    repeated TransactionReceipt receipts = 2;
```

CHAPTER

THREE

APPLICATION DEVELOPER'S GUIDE

This guide describes how to develop applications which run on top of the Hyperledger Sawtooth *core platform*, primarily through the use of Sawtooth's provided SDKs and *REST API*.

Topics include an overview of *Sawtooth transaction families*, setting up an application development environment, learning Sawtooth concepts with an example client, and tutorials on using the Sawtooth SDKs to write the business logic for your application.

Sawtooth provides SDKs in several languages, including Python, Javascript, Go, C++, Java, and Rust. This guide has tutorials for using the Go, Javascript, and Python SDKs.

3.1 Transaction Family Overview

Sawtooth separates the application level from the core system level with transaction families, which allows application developers to write in the languages of their choice. Each application defines the custom transaction families for its unique requirements.

A transaction family includes these components:

- A transaction processor to define the business logic for your application
- A data model to record and store data
- A client to handle the client logic for your application

See *Transaction Family Specifications* for a list of example transaction families. Sawtooth provides these examples to serve as models for low-level functions (such as maintaining chain-wide settings and storing on-chain permissions) and for specific applications such as performance analysis and storing block information.

3.2 Available SDKs

The following table summarizes the Sawtooth SDKs. It shows the feature completeness, API stability, and maturity level for the client signing, transaction processors, and state delta features.

	Client Signing			**Transaction Processor**			**State Delta**		
	Complete?	Stable API?	Maturity	Complete?	Stable API?	Maturity	Complete?	Stable API?	Maturity
Python	✓	✓	1	✓	✓	1	✓	✓	1
Go	✓	✓	1	✓	✓	1	✓	✓	1
JavaScript	✓	✓	1	✓	✓	2	✓	✓	2
Rust	✓		3			3		✓	3
Java			3			3			3
C++			3			3			3

A stable API means that the Sawtooth development team is committed to backward compatibility for future changes. Other APIs could change, which would require updates to application code.

The Maturity column shows the general maturity level of each feature:

1. Recommended: Well supported and heavily used
2. Community support only
3. Experimental: Might have known issues and future API changes

The available SDKs are in https://github.com/hyperledger/sawtooth-core/tree/master/sdk.

3.3 Installing and Running Sawtooth

Before you can start developing for the *Hyperledger Sawtooth* platform, you'll need to set up and run a local validator to test your application against. Once running, you will be able to submit new transactions and fetch the resulting state and block data from the blockchain using HTTP and the Sawtooth *REST API*. The methods detailed here will apply to the included example transaction families, *IntegerKey* and *XO*, as well as any transaction families you might write yourself.

Sawtooth validators can be run from prebuilt Docker containers, installed natively using Ubuntu 16.04 or launched in AWS from the AWS Marketplace. To get started, choose the platform appropriate to your use case and follow one of the installation and usage guides below.

3.3.1 Using Sawtooth with Docker

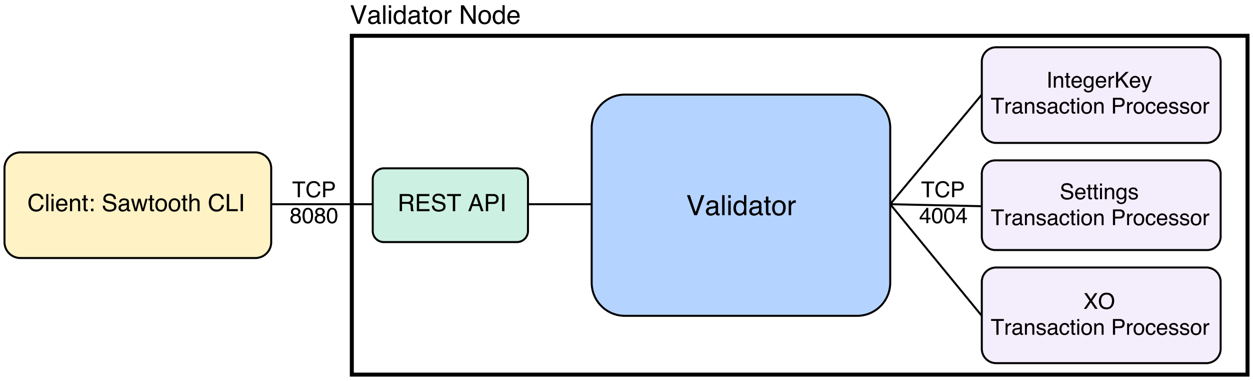
This procedure walks through the process of setting up Hyperledger Sawtooth for application development using Docker Compose, introduces some of the basic Sawtooth concepts necessary for application development, and walks through performing the following tasks:

- Submit transactions to the REST API
- View blocks, transactions, and state with Sawtooth commands
- Start and stop validators and transaction processors

Upon completing this tutorial, you will be prepared for the more advanced tutorials that guide you in performing app development tasks, such as implementing business logic with transaction families and writing clients which use Sawtooth's REST API.

About the Demo Sawtooth Environment

This tutorial describes how to use Docker to set up a single Sawtooth node with a validator, a REST API, and two transaction processors.



This demo environment introduces basic Sawtooth functionality with the IntegerKey and Settings transaction processors for the business logic and the Sawtooth CLI as the client. It also includes the XO transaction processor, which is used in a later tutorial.

Install Docker Engine and Docker Compose

Windows

Install the latest version of Docker Engine for Windows.

On Windows, Docker Compose is installed automatically when you install Docker Engine.

macOS

Install the latest version of Docker Engine for macOS.

On macOS, Docker Compose is installed automatically when you install Docker Engine.

Linux

On Linux, follow these steps:

1. Install Docker Engine.
2. Install Docker Compose.

Warning: Note that the minimum version of Docker Engine necessary is 17.03.0-ce. Linux distributions often ship with older versions of Docker.

Environment Setup

Download the Docker Compose File

A Docker Compose file is provided which defines the process for constructing a simple Sawtooth environment. This environment includes the following containers:

- A single validator using dev-mode consensus

- A REST API connected to the validator
- The Settings, IntegerKey, and XO transaction processors
- A client container for running Sawtooth commands

The Docker Compose file also specifies the container images to download from Docker Hub and the network settings needed for all the containers to communicate correctly.

This Docker Compose file can serve as the basis for your own multi-container Sawtooth development environment or application.

Download the Docker Compose file here.

Proxy Settings (Optional)

To configure Docker to work with an HTTP or HTTPS proxy server, follow the instructions for your operating system:

- Windows - See the instructions for proxy configuration in Get Started with Docker for Windows.
- macOS - See the instructions for proxy configuration in Get Started with Docker for Mac.
- Linux - See the instructions for proxy configuration in Control and configure Docker with Systemd.

Starting Sawtooth

To start up the environment, perform the following tasks:

1. Open a terminal window.
2. Change your working directory to the same directory where you saved the Docker Compose file.
3. Run the following command:

```
% docker-compose -f sawtooth-default.yaml up
```

Note: To learn more about the startup process, see *Using Sawtooth on Ubuntu 16.04*.

Downloading the Docker images that comprise the Sawtooth demo environment can take several minutes. Once you see the containers registering and creating initial blocks, you can move on to the next step.

```
Attaching to sawtooth-validator-default, sawtooth-xo-tp-python-default, sawtooth-
↪intkey-tp-python-default, sawtooth-rest-api-default, sawtooth-settings-tp-default,␣
↪sawtooth-shell-default
sawtooth-validator-default | writing file: /etc/sawtooth/keys/validator.priv
sawtooth-validator-default | writing file: /etc/sawtooth/keys/validator.pub
sawtooth-validator-default | creating key directory: /root/.sawtooth/keys
sawtooth-validator-default | writing file: /root/.sawtooth/keys/my_key.priv
sawtooth-validator-default | writing file: /root/.sawtooth/keys/my_key.pub
sawtooth-validator-default | Generated config-genesis.batch
sawtooth-validator-default | Processing config-genesis.batch...
sawtooth-validator-default | Generating /var/lib/sawtooth/genesis.batch
```

Stopping Sawtooth

If the environment needs to be reset or stopped for any reason, it can be returned to the default state by logging out of the client container, then pressing CTRL-c from the window where you originally ran `docker-compose`. Once the containers have all shut down, run `docker-compose -f sawtooth-default.yaml down`.

Sample output after pressing CTRL-c:

```
sawtooth-validator-default         | [00:27:56.753 DEBUG    interconnect] message␣
↪round trip: TP_PROCESS_RESPONSE 0.03986167907714844
sawtooth-validator-default         | [00:27:56.756 INFO     chain] on_block_
↪validated: 44ccc3e6(1, S:910b9c23, P:05b2a651)
sawtooth-validator-default         | [00:27:56.761 INFO     chain] Chain head updated␣
↪to: 44ccc3e6(1, S:910b9c23, P:05b2a651)
sawtooth-validator-default         | [00:27:56.762 INFO     publisher] Now building␣
↪on top of block: 44ccc3e6(1, S:910b9c23, P:05b2a651)
sawtooth-validator-default         | [00:27:56.763 INFO     chain] Finished block␣
↪validation of: 44ccc3e6(1, S:910b9c23, P:05b2a651)
Gracefully stopping... (press Ctrl+C again to force)
Stopping sawtooth-xo-tp-python-default ... done
Stopping sawtooth-settings-tp-default ... done
Stopping sawtooth-shell-default... done
Stopping sawtooth-rest-api-default ... done
Stopping sawtooth-intkey-tp-python-default ... done
Stopping sawtooth-validator-default ... done
```

After shutdown has completed, run this command:

```
% docker-compose -f sawtooth-default.yaml down
```

Logging Into The Client Container

The client container is used to run Sawtooth commands, which is the usual way to interact with validators or validator networks.

Log into the client container by opening a new terminal window and running the following command. Note that `sawtooth-shell-default` specifies the client container name.

```
% docker exec -it sawtooth-shell-default bash
```

Important: Your environment is ready for experimenting with Sawtooth. However, any work done in this environment will be lost once the container exits. The demo Docker Compose file is useful as a starting point for the creation of your own Docker-based development environment. In order to use it for app development, you would need to take additional steps, such as mounting a host directory into the container. See Docker's documentation for details.

Confirming Connectivity

To confirm that a validator is running and reachable from the client container, run this `curl` command as root:

```
root@75b380886502:/# curl http://rest-api:8008/blocks
```

To check connectivity from the host computer, open a new terminal window on your host system and use this `curl` command:

```
$ curl http://localhost:8008/blocks
```

If the validator is running and reachable, the output for each command should be similar to this example:

```
{
  "data": [
    {
      "batches": [],
      "header": {
        "batch_ids": [],
        "block_num": 0,
        "consensus": "R2VuZXNpcw==",
        "previous_block_id": "0000000000000000",
        "signer_public_key":
↪"03061436bef428626d11c17782f9e9bd8bea55ce767eb7349f633d4bfea4dd4ae9",
        "state_root_hash":
↪"708ca7fbb701799bb387f2e50deaca402e8502abe229f705693d2d4f350e1ad6"
      },
      "header_signature":
↪"119f076815af8b2c024b59998e2fab29b6ae6edf3e28b19de91302bd13662e6e43784263626b72b1c1ac120a491142ca25
↪"
    }
  ],
  "head":
↪"119f076815af8b2c024b59998e2fab29b6ae6edf3e28b19de91302bd13662e6e43784263626b72b1c1ac120a491142ca25
↪",
  "link": "http://rest-api:8008/blocks?
↪head=119f076815af8b2c024b59998e2fab29b6ae6edf3e28b19de91302bd13662e6e43784263626b72b1c1ac120a491142
↪",
  "paging": {
    "start_index": 0,
    "total_count": 1
  }
}root@75b380886502:/#
```

If the validator process or the validator container is not running, the `curl` command will time out or return nothing.

Using Sawtooth Commands

Creating and Submitting Transactions with intkey

The `intkey` command is provided to create sample transactions of the intkey (IntegerKey) transaction type for testing purposes. This step uses `intkey` to prepare batches of intkey transactions which set a few keys to random values, then randomly increment and decrement those values. These batches are saved locally, then submitted to the validator.

Run the following commands from the client container:

```
$ intkey create_batch --count 10 --key-count 5
$ intkey load -f batches.intkey -U http://rest-api:8008
```

The terminal window in which you ran the `docker-compose` command will begin logging output as the validator and IntegerKey transaction processor handle the transactions just submitted:

```
intkey-tp-python_1  | [21:02:53.164 DEBUG    handler] Incrementing "VaUEPt" by 1
sawtooth-validator-default          | [21:02:53.169 DEBUG    interconnect]␣
↪ServerThread receiving TP_STATE_SET_REQUEST message: 194 bytes
```

(continues on next page)

(continued from previous page)

```
sawtooth-validator-default         | [21:02:53.171 DEBUG    tp_state_handlers] SET: [
↪'1cf126d8a50604ea6ab1b82b33705fc3eeb7199f09ff2ccbc52016bbf33ade68dc23f5']
sawtooth-validator-default         | [21:02:53.172 DEBUG    interconnect]␣
↪ServerThread sending TP_STATE_SET_RESPONSE to b'63cf2e2566714070'
sawtooth-validator-default         | [21:02:53.176 DEBUG    interconnect]␣
↪ServerThread receiving TP_PROCESS_RESPONSE message: 69 bytes
sawtooth-validator-default         | [21:02:53.177 DEBUG    interconnect] message␣
↪round trip: TP_PROCESS_RESPONSE 0.042026519775390625
sawtooth-validator-default         | [21:02:53.182 DEBUG    interconnect]␣
↪ServerThread sending TP_PROCESS_REQUEST to b'63cf2e2566714070'
intkey-tp-python_1   | [21:02:53.185 DEBUG    core] received message of type: TP_
↪PROCESS_REQUEST
sawtooth-validator-default         | [21:02:53.191 DEBUG    interconnect]␣
↪ServerThread receiving TP_STATE_GET_REQUEST message: 177 bytes
sawtooth-validator-default         | [21:02:53.195 DEBUG    tp_state_handlers] GET: [(
↪'1cf126721fff0dc4ccb345fb145eb9e30cb7b046a7dd7b51bf7393998eb58d40df5f9a', b
↪'\xa1fZeuYwh\x1a\x00\x019%')]
sawtooth-validator-default         | [21:02:53.200 DEBUG    interconnect]␣
↪ServerThread sending TP_STATE_GET_RESPONSE to b'63cf2e2566714070'
intkey-tp-python_1   | [21:02:53.202 DEBUG    handler] Incrementing "ZeuYwh" by 1
```

Submitting Transactions with sawtooth batch submit

Instead of using `intkey load`, you can also submit transactions, including IntegerKey transactions, with the `sawtooth batch submit` command.

For example, you can submit the transactions in the file `batches.intkey` as generated above with this command:

```
$ sawtooth batch submit -f batches.intkey --url http://rest-api:8008
```

Viewing the Blockchain

You can view the blocks stored in the blockchain using the `sawtooth block` subcommand.

Note: The `sawtooth` command provides help for all subcommands. For example, to get help for the `block` subcommand, enter the command `sawtooth block -h`.

Viewing the List of Blocks

Enter the command `sawtooth block list` to view the blocks stored by the state:

```
$ sawtooth block list --url http://rest-api:8008
```

The output of the command will be similar to this:

```
NUM  BLOCK_ID                                                                                  ␣
↪                                                   BATS  TXNS  SIGNER
1    ␣
↪e8377628b299f4a3ff11ed173958205f30c1db12ea136ee75ab1b659d43dccd62bc994592faaca1d98d73044da902e9a8eb
↪ 11    63    027da204...
```

(continues on next page)

(continued from previous page)

```
0   ␣
↪97b210dd655fce913a76ec02c0fc131c8ec79b14592ec5170c55fea0c0c9fc7b6b84ba61d94d27d31a220d7301b33dd34c
↪ 1     1     027da204...
```

Viewing a Particular Block

From the output generated by the `sawtooth block list` command, copy the id of a block you want to get more info about, then paste it in place of `{BLOCK_ID}` in the following `sawtooth block show` command:

```
$ sawtooth block show --url http://rest-api:8008 {BLOCK_ID}
```

The output of this command includes all data stored under that block, and can be quite long. It should look something like this:

```
batches:
- header:
    signer_public_key:␣
↪0276023d4f7323103db8d8683a4b7bc1eae1f66fbbf79c20a51185f589e2d304ce
    transaction_ids:
    -␣
↪24b168aaf5ea4a76a6c316924a1c26df0878908682ea5740dd70814e7c400d56354dee788191be8e28393c70398906fb46
  header_signature:␣
↪a93731646a8fd2bce03b3a17bc2cb3192d8597da93ce735950dccbf0e3cf0b005468fadb94732e013be0bc2afb320be159
  transactions:
  - header:
      batcher_public_key:␣
↪0276023d4f7323103db8d8683a4b7bc1eae1f66fbbf79c20a51185f589e2d304ce
      dependencies: []
      family_name: sawtooth_settings
      family_version: '1.0'
      inputs:
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c1c0cbf0fbcaf64c0b
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c12840f169a04216b7
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c1918142591ba4e8a7
      - 000000a87cb5eafdcca6a8f82af32160bc531176b5001cb05e10bce3b0c44298fc1c14
      nonce: ''
      outputs:
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c1c0cbf0fbcaf64c0b
      - 000000a87cb5eafdcca6a8f82af32160bc531176b5001cb05e10bce3b0c44298fc1c14
      payload_sha512:␣
↪944b6b55e831a2ba37261d904b14b4e729399e4a7c41bd22fcb09c46f0b3821cd41750e38640e33f79b6b5745a20225a1f
      signer_public_key:␣
↪0276023d4f7323103db8d8683a4b7bc1eae1f66fbbf79c20a51185f589e2d304ce
    header_signature:␣
↪24b168aaf5ea4a76a6c316924a1c26df0878908682ea5740dd70814e7c400d56354dee788191be8e28393c70398906fb46
    payload:␣
↪EtwBCidzYXd0b290aC52YWxpZGF0b3IudHJhbnNhY3Rpb25fZmFtaWxpZXMSngFbeyJmYW1pbHkiOiAiaW50a2V5IiwgInZlcn
header:
  batch_ids:
  -␣
↪a93731646a8fd2bce03b3a17bc2cb3192d8597da93ce735950dccbf0e3cf0b005468fadb94732e013be0bc2afb320be159
  block_num: 3
  consensus: RGV2bW9kZQ==
  previous_block_id:␣
↪042f08e1ff49bbf16914a53dc9056fb6e522ca0e2cff872547eac9555c1de2a6200e67fb9daae6dfb90f02bef6a9088e94
```

(continues on next page)

(continued from previous page)

```
  signer_public_key:␣
↪033fbed13b51eafaca8d1a27abc0d4daf14aab8c0cbc1bb4735c01ff80d6581c52
  state_root_hash: 5d5ea37cbbf8fe793b6ea4c1ba6738f5eee8fc4c73cdca797736f5afeb41fbef
header_signature:␣
↪ff4f6705bf57e2a1498dc1b649cc9b6a4da2cc8367f1b70c02bc6e7f648a28b53b5f6ad7c2aa639673d873959f5d3fcc11
```

Viewing Global State

Viewing the List of Nodes (Addresses)

Use the command `sawtooth state list` to list the nodes in state (in the Merkle-Radix tree):

```
$ sawtooth state list --url http://rest-api:8008
```

The output of the command will be similar to this truncated list:

```
ADDRESS                                                                                     ␣
↪                                                                SIZE DATA
1cf126ddb507c936e4ee2ed07aa253c2f4e7487af3a0425f0dc7321f94be02950a081ab7058bf046c788dbaf0f10a980763e
↪11   b'\xa1fcCTdcH\x...
1cf1260cd1c2492b6e700d5ef65f136051251502e5d4579827dc303f7ed76ddb7185a19be0c6443503594c3734141d2bdcf5
↪11   b'\xa1frdLONu\x...
1cf126ed7d0ac4f755be5dd040e2dfcd71c616e697943f542682a2feb14d5f146538c643b19bcfc8c4554c9012e56209f94e
↪11   b'\xa1fAUZZqk\x...
1cf126c46ff13fcd55713bcfcf7b66eba515a51965e9afa8b4ff3743dc6713f4c40b4254df1a2265d64d58afa14a0051d3e3
↪11   b'\xa1fLvUYLk\x...
1cf126c4b1b09ebf28775b4923e5273c4c01ba89b961e6a9984632612ec9b5af82a0f7c8fc1a44b9ae33bb88f4ed39b590d4
↪13   b'\xa1fXHonWY\x...
1cf126e924a506fb2c4bb8d167d20f07d653de2447df2754de9eb61826176c7896205a17e363e457c36ccd2b7c124516a9b5
↪13   b'\xa1foWZXEz\x...
1cf126c295a476acf935cd65909ed5ead2ec0168f3ee761dc6f37ea9558fc4e32b71504bf0ad56342a6671db82cb8682d646
↪13   b'\xa1fadKGve\x...
```

Viewing Data at an Address

From the output generated by the `sawtooth state list` command, copy the address you want to view, then paste it in place of `{STATE_ADDRESS}` in the following `sawtooth state show` command:

```
$ sawtooth state show --url http://rest-api:8008 {STATE_ADDRESS}
```

The output of the command will include both the bytes stored at that address and the block id of the *chain head* the current state is tied to. It should look similar to this:

```
DATA: "b'\xa1fcCTdcH\x192B'"
HEAD:
↪"0c4364c6d5181282a1c7653038ec9515cb0530c6bfcb46f16e79b77cb524491676638339e8ff8e3cc57155c6d920e6a4d
↪"
```

Connecting to the REST API

From the Client Container

Use `curl` to confirm that you can connect to the REST API from the host. Enter the following command from the terminal window for the client container:

```
$ curl http://rest-api:8008/blocks
```

From the Host Operating System

Use `curl` to confirm that you can connect to the REST API from the host. Enter the following command from the terminal window for your host system:

```
$ curl http://localhost:8008/blocks
```

Connecting to Each Container

The Client Container

- Submits transactions
- Runs Sawtooth commands
- Container name: `sawtooth-shell-default`

No Sawtooth components are automatically started in this container.

Log into this container by running this command from the host computer's terminal window:

```
% docker exec -it sawtooth-shell-default bash
```

The Validator Container

- Runs a single validator
- Available to the other containers and host on TCP port 4004 (default)
- Hostname: `validator`
- Container name: `sawtooth-validator-default`

Log into this container by running this command from the host computer's terminal window:

```
$ docker exec -it sawtooth-validator-default bash
```

To see which components are running, use this command from the container:

```
$ ps --pid 1 fw
  PID TTY      STAT   TIME COMMAND
  1 ?        Ss     0:00 bash -c sawadm keygen && sawtooth keygen my_key && sawset␣
↪genesis -k /root/.sawtooth/keys/my_key.priv && sawadm genesis config-genesis.batch &
↪& sawtooth-validator -vv --endpoint
```

Note: The validator can process transactions in serial or parallel with no difference in the state produced. To process in parallel, give `sawtooth-validator` the option `--scheduler parallel` in the *sawtooth-default.yaml* file. The default option is `--scheduler serial`. To get the most benefit from the parallel option, add more transaction processors to the *sawtooth-default.yaml* file.

The REST API Container

- Runs the REST API
- Available to the client container and host on TCP port 8008
- Container name: `sawtooth-rest-api-default`

Log into this container by running this command from the host computer's terminal window:

```
$ docker exec -it sawtooth-rest-api-default bash
```

To see which components are running, run this command from the container:

```
$ ps --pid 1 fw
  PID TTY      STAT   TIME COMMAND
  1 ?        Ssl    0:02 /usr/bin/python3 /usr/bin/sawtooth-rest-api --connect tcp://
↪validator:4004 --bind rest-api:8008
```

The Settings Transaction Processor Container

- Runs a single Settings transaction processor
- Handles transactions of the Settings transaction family
- Hostname: `settings-tp`
- Container name: `sawtooth-settings-tp-default`

Log into this container by running this command from the host computer's terminal window:

```
$ docker exec -it sawtooth-settings-tp-default bash
```

To see which components are running, run this command from the container:

```
$ ps --pid 1 fw
  PID TTY      STAT   TIME COMMAND
  1 ?        Ssl    0:00 /usr/bin/python3 /usr/bin/settings-tp -vv tcp://
↪validator:4004
```

The IntegerKey Transaction Processor Container

- Runs a single IntegerKey transaction processor
- Handles transactions of the IntegerKey transaction family
- Hostname: `intkey-tp-python`
- Container name: `sawtooth-intkey-tp-python-default`

Log into this container by running this command from the host computer's terminal window:

```
$ docker exec -it sawtooth-intkey-tp-python-default bash
```

To see which components are running, run this command from the container:

```
$ ps --pid 1 fw
  PID TTY      STAT   TIME COMMAND
  1 ?        Ssl    0:00 /usr/bin/python3 /usr/bin/intkey-tp-python -vv tcp://
↪validator:4004
```

The XO Transaction Processor Container

- Runs a single XO transaction processor
- Handles transactions of the XO transaction family
- Hostname: `xo-tp-python`
- Container name: `sawtooth-xo-tp-python-default`

Log into this container by running this command from the host computer's terminal window:

```
$ docker exec -it sawtooth-xo-tp-python-default bash
```

To see which components are running, run this command from the container:

```
$ ps --pid 1 fw
  PID TTY      STAT   TIME COMMAND
  1 ?        Ssl    0:00 /usr/bin/python3 /usr/bin/xo-tp-python -vv tcp://
↪validator:4004
```

Viewing Log Files

You can view the log files for any running Docker container using the `docker logs` command. Replace `{CONTAINER}` with the name of one of the Sawtooth Docker containers, such as `sawtooth-validator-default`.

```
$ docker logs {CONTAINER}
```

Configuring the List of Transaction Families

Next, tell the validator or validator network to accept transactions from the IntegerKey and Settings transaction families.

Sawtooth provides a *Settings transaction family* that stores on-chain configuration settings, along with a Settings family transaction processor written in Python.

One of the on-chain settings is the list of supported transaction families. In the example below, a JSON array is submitted to the `sawset` command, which creates and submits a batch of transactions containing the configuration change.

The submitted JSON array tells the validator or validator network to accept transactions of the following types:

- intkey (IntegerKey transaction family)
- sawtooth_settings (Sawtooth transaction family)

To create and submit the batch containing the new setting, enter the following commands.

Note: The config command needs to use a key generated in the validator container. Thus, you must open a terminal window running in the validator container, rather than the client container (for the following command only). Run the following command on your host machine:

```
% docker exec -it sawtooth-validator-default bash
```

Then run the following command from the validator container:

```
$ sawset proposal create \
  --url http://rest-api:8008 \
  --key /root/.sawtooth/keys/my_key.priv \
  sawtooth.validator.transaction_families='[{"family": "intkey", "version": "1.0"}, {
↪"family":"sawtooth_settings", "version":"1.0"}]'
$ sawtooth settings list --url http://rest-api:8008
```

A TP_PROCESS_REQUEST message appears in the logging output of the validator, and output similar to the following appears in the validator terminal window:

```
sawtooth.settings.vote.authorized_keys:␣
↪0276023d4f7323103db8d8683a4b7bc1eae1f66fbbf79c20a51185f589e2d304ce
sawtooth.validator.transaction_families: [{"family": "intkey", "version": "1.0"}, {
↪"family":"sawtooth_settings", "versi...
```

3.3.2 Using Sawtooth on Ubuntu 16.04

This procedure guides you through the process of setting up Hyperledger Sawtooth for application development on Ubuntu, introduces some of the basic Sawtooth concepts necessary for application development, and walks through performing the following tasks:

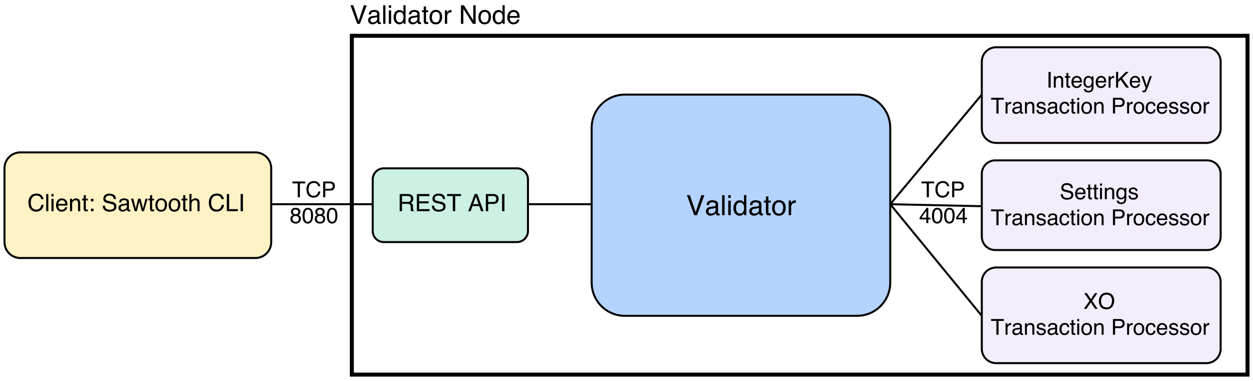
- Installing Sawtooth on Ubuntu 16.04
- Starting a Sawtooth validator and related components: the REST API and two transaction processors
- Configuring the transaction family settings
- Submitting transactions to the REST API
- Viewing blocks, transactions, and state with `sawtooth` commands

Upon completion of this section, you will be prepared for subsequent sections that describe application development topics, such as implementing business logic with transaction families and writing clients which use Sawtooth's REST API.

About the Demo Sawtooth Environment

This tutorial describes how to set up a single Sawtooth node with a validator, a REST API, and two transaction processors.

This demo environment introduces basic Sawtooth functionality with the IntegerKey and Settings transaction processors for the business logic and the Sawtooth CLI as the client. It also includes the XO transaction processor, which is used in a later tutorial.



Installing Sawtooth

Getting the Sawtooth Packages for Ubuntu

The Sawtooth package repositories provide two types of Ubuntu packages: stable or nightly. We recommend using the stable repository.

To add the stable repository, run these commands in a terminal window on your host system:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 8AA7AF1F1091A5FD
$ sudo add-apt-repository 'deb http://repo.sawtooth.me/ubuntu/1.0/stable xenial universe'
$ sudo apt-get update
```

To use the nightly repository, run the following commands in a terminal window on your host system:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 44FC67F19B2466EA
$ sudo apt-add-repository "deb http://repo.sawtooth.me/ubuntu/nightly xenial universe"
$ sudo apt-get update
```

Caution: Nightly builds may be out-of-sync with the documentation and have not gone through long-running network testing. We really do recommend the stable repository.

Installing the Sawtooth Packages

Sawtooth consists of several Ubuntu packages that can be installed together using the `sawtooth` metapackage. Run the following command in the same host terminal window:

```
$ sudo apt-get install -y sawtooth
```

At any time after installation, you can view the installed sawtooth packages with the following command:

```
$ dpkg -l '*sawtooth*'
```

Creating the Genesis Block

In most cases, it is not necessary to create a genesis block when starting a validator, because the validator joins an existing distributed ledger network. However, as a developer, you may often need to create short-lived test networks. In this case, you need to create a genesis block when instantiating a new network.

The genesis block contains some initial values that are necessary when a Sawtooth distributed ledger is created and used for the first time. One of the settings in the genesis block that should be set is the key that is authorized to set and change configuration settings, as shown below using the `sawset genesis` command.

To create the genesis block, open a new terminal window and run the following commands:

```
$ sawtooth keygen
$ sawset genesis
$ sudo -u sawtooth sawadm genesis config-genesis.batch
```

After the last command, the following output appears:

```
Processing config-genesis.batch...
Generating /var/lib/sawtooth/genesis.batch
```

Note: If you need to delete previously existing blockchain data before running a validator, remove all files from `/var/lib/sawtooth`.

Starting the Validator

To start a validator that listens locally on the default ports, run the following commands:

```
$ sudo sawadm keygen
$ sudo -u sawtooth sawtooth-validator -vv
```

Logging output is displayed in the validator terminal window. The output ends with lines similar to these:

```
[2017-12-05 22:33:42.785 INFO     chain] Chain controller initialized with chain␣
↪head: c788bbaf(2, S:3073f964, P:c37b0b9a)
[2017-12-05 22:33:42.785 INFO     publisher] Now building on top of block: c788bbaf(2,
↪ S:3073f964, P:c37b0b9a)
[2017-12-05 22:33:42.788 DEBUG    publisher] Loaded batch injectors: []
[2017-12-05 22:33:42.866 DEBUG    interconnect] ServerThread receiving TP_REGISTER_
↪REQUEST message: 92 bytes
[2017-12-05 22:33:42.866 DEBUG    interconnect] ServerThread receiving TP_REGISTER_
↪REQUEST message: 103 bytes
[2017-12-05 22:33:42.867 INFO     processor_handlers] registered transaction␣
↪processor: connection_
↪id=4c2d581131c7a5213b4e4da63180048ffd8983f6aa82a380ca28507bd3a96d40027a797c2ee59d029e42b7b1b4cc470
↪ family=intkey, version=1.0, namespaces=['1cf126']
[2017-12-05 22:33:42.867 DEBUG    interconnect] ServerThread sending TP_REGISTER_
↪RESPONSE to b'c61272152064480f'
[2017-12-05 22:33:42.869 INFO     processor_handlers] registered transaction␣
↪processor: connection_
↪id=e80eb89943398f296b1c99e45b5b31a9647d1c15a412842c804222dcc0e3f3a3045b6947bab06f42c5f79acdcde91be
↪ family=sawtooth_settings, version=1.0, namespaces=['000000']
[2017-12-05 22:33:42.869 DEBUG    interconnect] ServerThread sending TP_REGISTER_
↪RESPONSE to b'a85335fced9b496e'
```

Tip: If you want to stop the validator, enter CTRL-c several times in the validator's terminal window.

Other Start-up Options for the Validator

In the `sawtooth-validator` command above, the *-vv* flag sets the log level to include DEBUG messages. To run the validator with less logging output, use *-v* or omit the flag.

By default, the validator listens on the loopback interface for both network and component communications. To change the interface and port used, the *–bind* flag can be used. See *sawtooth-validator* for more information on the available flags. The following command is equivalent to the default behavior:

```
sudo -u sawtooth sawtooth-validator -vv --bind network:tcp://127.0.0.1:8800 --bind component:tcp://127.0.0.1:4004
```

The validator can process transactions in serial (the default) or parallel with no difference in the state produced. To process in parallel, use the option `--scheduler parallel`. To get the most benefit from the parallel option, start multiple transaction processors for the types of transactions where there is an expected high volume.

Starting the REST API

In order to configure a running validator, submit batches, and query the state of the ledger, you must start the REST API application.

Open a new terminal window, then run the following command to start the REST API and connect to a local validator:

```
$ sudo -u sawtooth sawtooth-rest-api -v
```

Starting and Configuring the Transaction Processors

This section describes how to start the IntegerKey and Settings family transaction processors, confirm that the REST API is running, and tell the validator or validator network to accept transactions from these transaction families.

Transaction processors can be started either before or after the validator is started.

Starting the IntegerKey Transaction Processor

The IntegerKey transaction processor is provided as a simple example of a transaction family, which can also be used for testing purposes.

To start an IntegerKey transaction processor, open a new terminal window, then run the following command:

```
$ sudo -u sawtooth intkey-tp-python -v
```

Note: By default, the transaction processor tries to connect to a local validator on port 4004. This can be modified by passing a different endpoint as an argument. The following endpoint argument is equivalent to the default: `intkey-tp-python -v tcp://127.0.0.1:4004`

This command starts a transaction processor with an `intkey` handler that can understand and process transactions from the IntegerKey transaction family.

The transaction processor produces the following output:

```
[23:07:57 INFO    core] register attempt: OK
```

Starting the Settings Family Transaction Processor

Sawtooth provides a *Settings transaction family* that stores on-chain settings, along with a Settings family transaction processor written in Python.

To start the Settings family transaction processor, open a new terminal window and run the following command:

```
$ sudo -u sawtooth settings-tp -v
```

Check the validator window to confirm that the transaction processor has registered with the validator. A successful registration event produces the following output:

```
[21:03:55.955 INFO    processor_handlers] registered transaction processor: identity=b
↪'6d2d80275ae280ea', family=sawtooth_settings, version=1.0, namespaces=<google.
↪protobuf.pyext._message.RepeatedScalarContainer object at 0x7e1ff042f6c0>
[21:03:55.956 DEBUG   interconnect] ServerThread sending TP_REGISTER_RESPONSE to b
↪'6d2d80275ae280ea'
```

Verifying that the REST API is Running

In order to configure a running validator, the REST API must be running. Open a new terminal window and run the following command:

```
$ ps aux | grep sawtooth-rest-api
sawtooth  2829  0.0  0.3  55756  3980 pts/0    S+   19:36   0:00 sudo -u sawtooth␣
↪sawtooth-rest-api -v
sawtooth  2830  0.0  3.6 221164 37520 pts/0    Sl+  19:36   0:00 /usr/bin/python3 /
↪usr/bin/sawtooth-rest-api -v
ubuntu    3004  0.0  0.0  12944   928 pts/4    S+   19:54   0:00 grep -E --color=auto␣
↪sawtooth-rest-api
```

If necessary, run the following command to start the REST API.

```
$ sudo -u sawtooth sawtooth-rest-api -v
```

Configuring the List of Transaction Families (Ubuntu version)

One of the on-chain settings is the list of supported transaction families. To configure this setting, use the following steps to create and submit the batch to change the transaction family settings.

In this example, the `sawset` command specifies a JSON array which creates and submits a batch of transactions containing the settings change. This JSON array tells the validator or validator network to accept transactions of the following types:

- `intkey` (IntegerKey transaction family)
- `sawtooth_settings` (Settings transaction family)

To create and submit the batch containing the new settings, enter the following command:

```
$ sawset proposal create sawtooth.validator.transaction_families='[{"family": "intkey
↪", "version": "1.0"}, {"family":"sawtooth_settings", "version":"1.0"}]'
```

The output in the validator terminal window includes TP_PROCESS_REQUEST messages and information on the authorized keys and transaction families, as in this truncated example:

```
...
[2017-12-05 22:42:46.269 DEBUG    tp_state_handlers] GET: [('000000a87c...\n&sawtooth.
↪settings.vote.authorized_keys\x12B0251fd...
...
[2017-12-05 22:42:46.274 DEBUG    tp_state_handlers] GET: [('000000a87c...'sa    ␣
↪wtooth.validator.transaction_families\x12Y[{"family": "intkey", "version": "    1.0
↪"}, {"family":"sawtooth_settings", "version":"1.0"}]')]
...
[2017-12-05 22:52:33.495 DEBUG    interconnect] ServerThread sending TP_PROCESS_
↪REQUEST to b'1893abb39b4b4aae'
...
```

Creating and Submitting Transactions

The `intkey` command creates sample `intkey` (IntegerKey) transactions for testing purposes.

This section guides you through the following tasks:

1. Preparing a batch of `intkey` transactions that set the keys to random values.
2. Generating *inc* (increment) and *dec* (decrement) transactions to apply to the existing state stored in the blockchain.
3. Submitting these transactions to the validator.

Run the following commands:

```
$ intkey create_batch
Writing to batches.intkey...

$ intkey load
batches: 2 batch/sec: 135.96900883377907
```

You can watch the processing of the transactions by observing the logging output of the `intkey` transaction processor. A truncated example of this output is shown below:

```
[19:29:26 INFO    core] register attempt: OK
[19:31:06 INFO    handler] processing: Verb=set Name=eBuPof Value=99811␣
↪address=1cf126c584128aaf1837c90c83748ab222c11b8bbd2fe6cc30f17fe35f2acb9af8efd4ee3f092b676546316cf85
[19:31:06 INFO    handler] processing: Verb=set Name=HOUUQS Value=10140␣
↪address=1cf126380fa9e716a05ac815741fd1960d5952e60f8747e13334f79504c57d0287b77cf9b78284d0e1544f6f03
[19:31:06 INFO    handler] processing: Verb=set Name=lrnuDC Value=92318␣
↪address=1cf12617c797cf8c27254bbdb5c9bda09f9405b9494ae32b79b9b6d30881ca8552d5932a68f703d1b6754b9feb
[19:31:06 INFO    handler] processing: Verb=set Name=BKaiql Value=94175␣
↪address=1cf12669cbc17d076a1accb4b0bb61f40ed4f999173b90e3ca2591875a55fee2947661e60fa1c57b41ef0f2660
[19:31:06 INFO    handler] processing: Verb=set Name=wpMQmE Value=47316␣
↪address=1cf1260f6bdf66b65ff7c00ec58c4deccffd167bfee7a85698880dfa485df3de1ec18a5b2d1dc12849743d1c74
[19:31:06 INFO    handler] processing: Verb=set Name=GTgrvP Value=31921␣
↪address=1cf12606ac7db03c756133c07d7d02b59f3ef9eae6774fe59c75c88ab66a9fabbbaef9975dbf9aa197d1090ed1
```

Viewing Blocks and State

You can view the blocks stored in the blockchain and the nodes of the Merkle-Radix tree (global state) by using the `sawtooth block` command.

Note: The `sawtooth` command provides help for all subcommands. For example, to get help for the `block` subcommand, enter the command `sawtooth block -h`.

Viewing the List of Blocks

Enter the command `sawtooth block list` to view the blocks stored by the state:

```
$ sawtooth block list
```

Tip: Expand the terminal window to at least 157 characters to view all output.

The output of the command will be similar to this:

```
NUM  BLOCK_ID                                                                                ␣
↪                                                  BATS   TXNS   SIGNER
2    ␣
↪5d4b9ba0c9b0615fc21fa89fe88c20fc3d2e2dba02e4b5e0df15ace9283dc4c62bc7b222d897c784ec1e4cc7737598376355
↪ 2      10     038b5e...
1    ␣
↪05d2f2101d30c7d9cc31b8f416818acf55a283828fcd45052fd51359e89c3a6a60c6f87354e2000e759754bf829bd375d27
↪ 1      1      038b5e...
0    ␣
↪8b2781db2a19936d8873e7e0a44c5294ea1ad110984d565c3fb669169dfd3514790405634a27225f6ab0cfd0301c434adfa
↪ 1      1      038b5e...
```

Viewing a Particular Block

Using the output from the `sawtooth block list` above, copy the block id you want to view, then paste it in place of `{BLOCK_ID}` in the following `sawtooth block show` command:

```
$ sawtooth block show {BLOCK_ID}
```

The output of this command will be similar to this example (truncated output shown):

```
  batches:
- header:
    signer_public_key:␣
↪0380be3421629849b1d03af520d7fa2cdc24c2d2611771ddf946ef3aaae216be84
    transaction_ids:
    -␣
↪c498c916da09450597053ada1938858a11d94e2ed5c18f92cd7d34b865af646144d180bdc121a48eb753b4abd326baa3ea2
    -␣
↪c68de164421bbcfcc9ea60b725bae289aecd02ddde6f520e6e85b3227337e2971e89bbff468bdebe408e0facc343c612a32
    -␣
↪faf9121f9744716363253cb0ff4b6011093ada6e19dae63ae04a58a1fca25424779a13628a047c009d2e73d0e7baddc95b4
  header_signature:␣
↪2ff874edfa80a8e6b718e7d10e91970150fcc3fcfd46d38eb18f356e7a733baa40d9e81
```

(continues on next page)

(continued from previous page)

```
 transactions:
 - header:
     batcher_public_key:␣
↪0380be3421629849b1d03af520d7fa2cdc24c2d2611771ddf946ef3aaae216be84
     dependencies:
     -␣
↪19ad647bd292c980e00f05eed6078b471ca2d603b842bc4eaecf301d61f15c0d3705a4ec8d915ceb646f35d443da43569f5
     family_name: intkey
     family_version: '1.0'
     inputs:
     -␣
↪1cf126c15b04cb20206d45c4d0e432d036420401dbd90f064683399fae55b99af1a543f7de79cfafa4f220a22fa248f834
     nonce: 0x1.63021cad39ceep+30
     outputs:
     -␣
↪1cf126c15b04cb20206d45c4d0e432d036420401dbd90f064683399fae55b99af1a543f7de79cfafa4f220a22fa248f834
     payload_sha512:␣
↪942a09c0254c4a5712ffd152dc6218fc5453451726d935ac1ba67de93147b5e7be605da7ab91245f48029b41f493a1cc8d
     signer_public_key:␣
↪0380be3421629849b1d03af520d7fa2cdc24c2d2611771ddf946ef3aaae216be84
   header_signature:␣
↪c498c916da09450597053ada1938858a11d94e2ed5c18f92cd7d34b865af646144d180bdc121a48eb753b4abd326baa3ea
   payload: o2ROYW1lZnFrbGR1emVWYWx1ZQFkVmVyYmNpbmM=
```

Viewing Global State

Use the command `sawtooth state list` to list the nodes in state (in the Merkle-Radix tree):

```
$ sawtooth state list
```

The output of the command will be similar to this truncated list:

```
ADDRESS                                                                                            ␣
↪                                                   SIZE DATA
1cf126ddb507c936e4ee2ed07aa253c2f4e7487af3a0425f0dc7321f94be02950a081ab7058bf046c788dbaf0f10a980763e
↪11   b'\xa1fcCTdcH\x...
1cf1260cd1c2492b6e700d5ef65f136051251502e5d4579827dc303f7ed76ddb7185a19be0c6443503594c3734141d2bdcf5
↪11   b'\xa1frdLONu\x...
1cf126ed7d0ac4f755be5dd040e2dfcd71c616e697943f542682a2feb14d5f146538c643b19bcfc8c4554c9012e56209f94e
↪11   b'\xa1fAUZZqk\x...
1cf126c46ff13fcd55713bcfcf7b66eba515a51965e9afa8b4ff3743dc6713f4c40b4254df1a2265d64d58afa14a0051d3e3
↪11   b'\xa1fLvUYLk\x...
1cf126c4b1b09ebf28775b4923e5273c4c01ba89b961e6a9984632612ec9b5af82a0f7c8fc1a44b9ae33bb88f4ed39b590d4
↪13   b'\xa1fXHonWY\x...
1cf126e924a506fb2c4bb8d167d20f07d653de2447df2754de9eb61826176c7896205a17e363e457c36ccd2b7c124516a9b5
↪13   b'\xa1foWZXEz\x...
1cf126c295a476acf935cd65909ed5ead2ec0168f3ee761dc6f37ea9558fc4e32b71504bf0ad56342a6671db82cb8682d646
↪13   b'\xa1fadKGve\x...
```

Note: An address is equivalent to a node id.

Viewing Data in a Node

You can use `sawtooth state show` command to view data for a specific node. Using the output from the `sawtooth state list` command above, copy the node id you want to view, then paste it in place of `{NODE_ID}` in the following command:

```
$ sawtooth state show {NODE_ID}
```

The output of the command will be similar to this:

```
DATA: "b'\xa1fcCTdcH\x192B'"
HEAD:
↪"0c4364c6d5181282a1c7653038ec9515cb0530c6bfcb46f16e79b77cb524491676638339e8ff8e3cc57155c6d920e6a4d
↪"
```

Stopping Sawtooth Components

To stop the validator, enter CTRL-c several times in the validator's terminal window.

Stop the REST API and transaction processors by entering a single CTRL-c in the appropriate windows.

3.3.3 Using AWS for Your Development Environment

This tutorial describes how to set up Hyperledger Sawtooth for application development using the Amazon Elastic Compute Cloud (Amazon EC2) service. This step-by-step procedure shows you how to launch an instance of a Sawtooth validator node from the Amazon Web Services (AWS) Marketplace, then walks you through the following tasks:

- Checking the status of Sawtooth components
- Configuring transaction family settings
- Submitting transactions to the REST API
- Viewing blocks and state with Sawtooth commands
- Examining Sawtooth logs
- Resetting the AWS Sawtooth instance

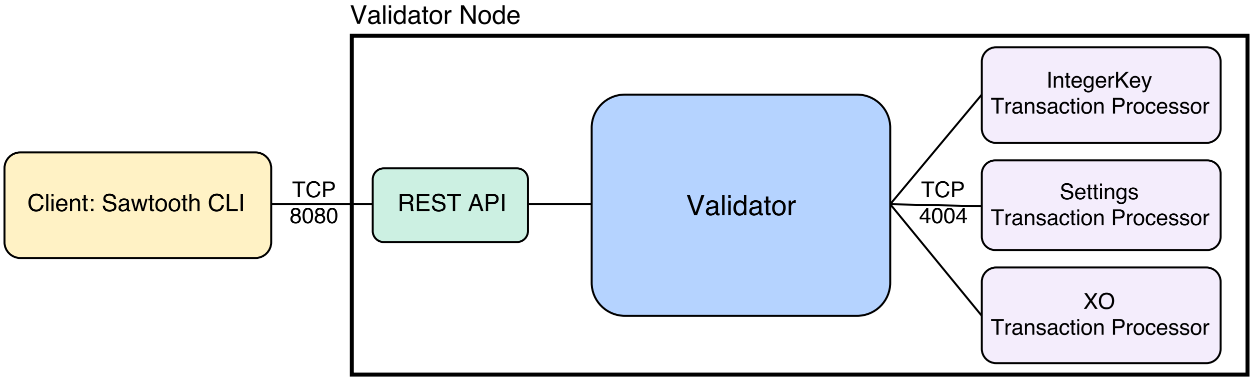
Note: This environment requires an AWS account. If you don't have one yet, Amazon provides free accounts so you can try the Sawtooth platform. Sign up at aws.amazon.com/free.

After completing this tutorial, you will be prepared for the advanced tutorials on creating a custom transaction family to implement your business logic and writing a client application that uses the Sawtooth REST API.

About the Demo Sawtooth Environment

The AWS demo environment is a single validator node that is running a validator, a REST API, and three transaction processors.

This demo environment introduces basic Sawtooth functionality with the IntegerKey and Settings transaction processors for the business logic and the Sawtooth CLI as the client. It also includes the XO transaction processor, which is used in the advanced tutorials.



The Amazon Machine Image (AMI) for Sawtooth has a `systemd` service that handles environment setup steps such as generating keys and creating a genesis block. To learn how the typical startup process works, see *Using Sawtooth on Ubuntu 16.04*.

Launching a Sawtooth Instance

1. Launch a Sawtooth instance from the Hyperledger Sawtooth product page on the AWS Marketplace.

   For more information, see the Amazon guide Launching an AWS Marketplace Instance.

   Note: The default security group with recommended settings for Sawtooth allows inbound SSH traffic only.

   - To attach a transaction processor remotely, add an inbound rule to allow TCP traffic on port 4004.
   - To access the REST API remotely, add an inbound rule to allow TCP traffic on port 8008.
   - To communicate with another validator node, add an inbound rule to allow TCP traffic on port 8800.

   For information on editing the security group rules, see Amazon's Security Groups documentation, Adding, Removing, and Updating Rules.

2. Log into this Sawtooth instance. Use the user name `ubuntu` when connecting.

   For more information, see the Amazon guide Connect to Your Linux Instance.

Once launched, the Sawtooth instance continues to run until you stop it. If you're uncertain about the state or would like to start over, see *Resetting the Environment*.

Checking the Status of Sawtooth Components

1. You can use `ps` to check that each Sawtooth component is running:

```
$ ps aux | grep [s]awtooth
sawtooth 27556 32.3  2.5 5371560864 407176 ?   Ssl  15:15  15:53 /usr/bin/
↪python3 /usr/bin/sawtooth-validator
sawtooth 27592  1.1  0.2 305796 38816 ?        Ssl  15:15   0:33 /usr/bin/
↪python3 /usr/bin/sawtooth-rest-api
sawtooth 27622  0.0  0.2 278176 33708 ?        Ssl  15:15   0:00 /usr/bin/
↪python3 /usr/bin/identity-tp -v -C tcp://localhost:4004
sawtooth 27712  0.0  0.2 278172 33632 ?        Ssl  15:15   0:00 /usr/bin/
↪python3 /usr/bin/settings-tp -C tcp://localhost:4004
```

2. Or you can use `systemctl`:

   ```
   $ systemctl | grep [s]awtooth
   sawtooth-identity-tp.service
        loaded active running   Sawtooth TP Identity
   sawtooth-rest-api.service
        loaded active running   Sawtooth REST API
   sawtooth-settings-tp.service
        loaded active running   Sawtooth TP Settings
   sawtooth-validator.service
        loaded active running   Sawtooth Validator Server
   ```

3. Confirm that you can connect to the REST API from your host system. Enter the following `curl` command from a terminal window:

```
$ curl http://localhost:8008/blocks
```

Configuring Transaction Family Settings

By default, Sawtooth allows transactions from any transaction family. In this step, you will change the configuration settings to allow only IntegerKey and Settings transactions.

This configuration change is itself a transaction that is submitted in a batch. Sawtooth configuration settings are stored on the blockchain so that all validator nodes in a Sawtooth network know the configuration.

1. The `sawset proposal` command creates and submits a batch containing the new settings. This batch submits a JSON array that tells the validator to accept only transactions of specified transaction type (in this case, `intkey` and `sawtooth_settings`).

   Enter the following command:

```
$ sawset proposal create sawtooth.validator.transaction_families='[{"family":
↪"intkey", "version": "1.0"}, {"family":"sawtooth_settings", "version":"1.0"}]'
```

   A `TP_PROCESS_REQUEST` message appears in the logging output of the validator, and output similar to the following appears in the `validator-debug.log` file in `/var/log/sawtooth`:

```
[21:11:55.356 [Thread-9] tp_state_handlers DEBUG] GET: [(
↪'000000a87cb5eafdcca6a8cde0fb0dec1400c5ab274474a6aa82c12840f169a04216b7',b
↪'\nl\n&sawtooth.settings.vote.authorized_
↪keys\x12B03e3ccf73dd618ef1abe18da84d3cf5838a5d292d36ef8857a60b5ad04fd4ab517')]
[21:11:55.356 [Thread-9] interconnect DEBUG] ServerThread sending TP_STATE_GET_
↪RESPONSE to b'afb61daaa87a4c70'
[21:11:55.362 [InterconnectThread-1] interconnect DEBUG] ServerThread receiving␣
↪TP_STATE_GET_REQUEST message: 177 bytes
[21:11:55.371 [InterconnectThread-1] interconnect DEBUG] message round trip: TP_
↪PROCESS_RESPONSE 0.021718978881835938
[21:11:55.373 [Thread-23] chain INFO] on_block_validated: 5da8c003(12, S:eb09cdf9,
↪ P:dab828cd)
[21:11:55.374 [Thread-23] chain INFO] Chain head updated to: 5da8c003(12,␣
↪S:eb09cdf9, P:dab828cd)
[21:11:55.374 [Thread-23] publisher INFO] Now building on top of block:␣
↪5da8c003(12, S:eb09cdf9, P:dab828cd)
[21:11:55.375 [Thread-23] chain DEBUG] Verify descendant blocks: 5da8c003(12,␣
↪S:eb09cdf9, P:dab828cd) ([])
[21:11:55.375 [Thread-23] state_delta_processor DEBUG] Publishing state delta␣
↪from 5da8c003(12, S:eb09cdf9, P:dab828cd)
```

(continues on next page)

(continued from previous page)

```
[21:11:55.376 [Thread-23] chain INFO] Finished block validation of: 5da8c003(12,␣
↪S:eb09cdf9, P:dab828cd)
```

2. Use `sawtooth settings list` to verify that the change was successfully applied.

```
$ sawtooth settings list
sawtooth.settings.vote.authorized_keys:␣
↪03e3ccf73dd618ef1abe18da84d3cf5838a5d292d36ef8857a60b5ad04fd4ab517
sawtooth.validator.transaction_families: [{"family": "intkey", "version": "1.0"},
↪{"family":"sawtooth_settings", "version":"1.0"} "...
```

Creating and Submitting Transactions

Sawtooth includes commands that act as a client application. This section describes how to use the `intkey` and `sawtooth batch` commands to create and submit transactions.

Note: Use the `--help` flag with any Sawtooth command to display the available options and subcommands.

Using intkey to Create and Submit Transactions

The `intkey` command creates and submits IntegerKey transactions for testing purposes.

1. Use `intkey create_batch` to prepare batches of transactions that set a few keys to random values, then randomly increment and decrement those values. These batches are saved locally in the file `batches.intkey`.

```
$ intkey create_batch --count 10 --key-count 5
Writing to batches.intkey...
```

2. Use `intkey load` to submit the batches to the validator.

```
$ intkey load
batches: 11 batch/sec: 141.7800162868952
```

3. This output doesn't tell us much, so let's look at the Sawtooth logs to see what happened. Examine the log files `validator-*-debug.log` and `intkey-*-debug.log` in the directory `/var/log/sawtooth`.

Note: Sawtooth log files contain a string to make the file names unique. The file names on your system may be different than the examples below.

4. Display the last 10 entries in the validator log. These entries show that state is being updated and that a new block has been published.

```
$ tail -10 /var/log/sawtooth/validator-d1cf3f4ffff81f50-debug.log
[20:52:07.835 [Thread-8] tp_state_handlers DEBUG] SET: [
↪'1cf1263d536e5febddb1d9804041192faea99c5cd784788a1e3e444d2db93ba60baa08']
[20:52:07.836 [Thread-8] interconnect DEBUG] ServerThread sending TP_STATE_SET_
↪RESPONSE to b'ae98c3726f9743c4'
[20:52:07.837 [InterconnectThread-1] interconnect DEBUG] ServerThread receiving␣
↪TP_PROCESS_RESPONSE message: 69 bytes
```

(continues on next page)

(continued from previous page)

```
[20:52:07.837 [InterconnectThread-1] interconnect DEBUG] message round trip: TP_
↪PROCESS_RESPONSE 0.006524801254272461
[20:52:07.843 [Thread-23] chain INFO] on_block_validated: a2ea3764(5, S:8e87d579,␣
↪P:62f7c965)
[20:52:07.844 [Thread-23] chain INFO] Chain head updated to: a2ea3764(5,␣
↪S:8e87d579, P:62f7c965)
[20:52:07.844 [Thread-23] publisher INFO] Now building on top of block:␣
↪a2ea3764(5, S:8e87d579, P:62f7c965)
[20:52:07.845 [Thread-23] chain DEBUG] Verify descendant blocks: a2ea3764(5,␣
↪S:8e87d579, P:62f7c965) ([])
[20:52:07.845 [Thread-23] state_delta_processor DEBUG] Publishing state delta␣
↪from a2ea3764(5, S:8e87d579, P:62f7c965)
[20:52:07.846 [Thread-23] chain INFO] Finished block validation of: a2ea3764(5,␣
↪S:8e87d579, P:62f7c965)
```

5. Display the last 10 entries in the intkey log. These entries show that values are being incremented and decremented.

```
$ tail -10 /var/log/sawtooth/intkey-ae98c3726f9743c4-debug.log
[20:52:07.803 [MainThread] core DEBUG] received message of type: TP_PROCESS_
↪REQUEST
[20:52:07.805 [MainThread] handler DEBUG] Decrementing "zhUyYM" by 6
[20:52:07.810 [MainThread] core DEBUG] received message of type: TP_PROCESS_
↪REQUEST
[20:52:07.812 [MainThread] handler DEBUG] Incrementing "ARqIDG" by 8
[20:52:07.817 [MainThread] core DEBUG] received message of type: TP_PROCESS_
↪REQUEST
[20:52:07.820 [MainThread] handler DEBUG] Decrementing "FxVRRq" by 6
[20:52:07.824 [MainThread] core DEBUG] received message of type: TP_PROCESS_
↪REQUEST
[20:52:07.827 [MainThread] handler DEBUG] Incrementing "hTnaor" by 9
[20:52:07.832 [MainThread] core DEBUG] received message of type: TP_PROCESS_
↪REQUEST
[20:52:07.834 [MainThread] handler DEBUG] Incrementing "ARqIDG" by 6
```

Using sawtooth batch submit to Submit Transactions

In the example above, the `intkey create_batch` command created the file `batches.intkey`. Rather than using `intkey load` to submit these transactions, you could use the following command to submit them:

```
$ sawtooth batch submit -f batches.intkey
```

Viewing Blockchain and Block Data

The `sawtooth block` command displays information about the blocks stored on the blockchain.

1. Use `sawtooth block list` to display the list of blocks stored in state.

```
$ sawtooth block list
```

The output shows the block number and block ID, as in this example:

```
NUM  BLOCK_ID
8    ␣
↪22e79778855768ea380537fb13ad210b84ca5dd1cdd555db7792a9d029113b0a183d5d71cc5558e04d10a9a9d49031
↪ 2     8     02a0e049...
7    ␣
↪c84346f5e18c6ce29f1b3e6e31534da7cd538533457768f86a267053ddf73c4f1139c9055be283dfe085c94557de24
↪ 2     20    02a0e049...
6    ␣
↪efc0d6175b6329ac5d0814546190976bc6c4e18bd0630824c91e9826f93c7735371f4565a8e84c706737d360873fac
↪ 2     27    02a0e049...
5    ␣
↪840c0ef13023f93e853a4555e5b46e761fc822d4e2d9131581fdabe5cb85f13e2fb45a0afd5f5529fbde5216d22a88
↪ 2     16    02a0e049...
4    ␣
↪4d6e0467431a409185e102301b8bdcbdb9a2b177de99ae139315d9b0fe5e27aa3bd43bda6b168f3ac8f45e84b06929
↪ 2     20    02a0e049...
3    ␣
↪9743e39eadf20e922e242f607d847445aba18dacdf03170bf71e427046a605744c84d9cb7d440d257c21d11e4da47e
↪ 2     22    02a0e049...
2    ␣
↪6d7e641232649da9b3c23413a31db09ebec7c66f8207a39c6dfcb21392b033163500d367f8592b476e0b9c1e621d6c
↪ 2     38    02a0e049...
1    ␣
↪7252a5ab3440ee332aef5830b132cf9dc3883180fb086b2a50f62bf7c6c8ff08311b8009da3b3f6e38d3cfac1b3ac4
↪ 2     120   02a0e049...
0    ␣
↪8821a997796f3e38a28dbb8e418ed5cbdd60b8a2e013edd20bca7ebf9a58f1302740374d98db76137e48b41dc404de
↪ 0     0     02a0e049...
```

2. Use `sawtooth block show` to view a specific block. Copy the block ID from the output of `sawtooth block list`, then specify this ID in place of `{BLOCK_ID}` in the following command.

```
$ sawtooth block show {BLOCK_ID}
```

The output of this command can be quite long, because it includes all data stored under that block. For example:

```
batches:
- header:
    signer_public_key:␣
↪0276023d4f7323103db8d8683a4b7bc1eae1f66fbbf79c20a51185f589e2d304ce
    transaction_ids:
    -␣
↪24b168aaf5ea4a76a6c316924a1c26df0878908682ea5740dd70814e7c400d56354dee788191be8e28393c70398906
  header_signature:␣
↪a93731646a8fd2bce03b3a17bc2cb3192d8597da93ce735950dccbf0e3cf0b005468fadb94732e013be0bc2afb320b
  transactions:
  - header:
      batcher_public_key:␣
↪0276023d4f7323103db8d8683a4b7bc1eae1f66fbbf79c20a51185f589e2d304ce
      dependencies: []
      family_name: sawtooth_settings
      family_version: '1.0'
      inputs:
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c1c0cbf0fbcaf64c0b
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c12840f169a04216b7
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c1918142591ba4e8a7
      - 000000a87cb5eafdcca6a8f82af32160bc531176b5001cb05e10bce3b0c44298fc1c14
```

(continues on next page)

(continued from previous page)

```
      nonce: ''
      outputs:
      - 000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c1c0cbf0fbcaf64c0b
      - 000000a87cb5eafdcca6a8f82af32160bc531176b5001cb05e10bce3b0c44298fc1c14
      payload_sha512:␣
↪944b6b55e831a2ba37261d904b14b4e729399e4a7c41bd22fcb09c46f0b3821cd41750e38640e33f79b6b5745a2022
      signer_public_key:␣
↪0276023d4f7323103db8d8683a4b7bc1eae1f66fbbf79c20a51185f589e2d304ce
    header_signature:␣
↪24b168aaf5ea4a76a6c316924a1c26df0878908682ea5740dd70814e7c400d56354dee788191be8e28393c70398906
    payload:␣
↪EtwBCidzYXd0b290aC52YWxpZGF0b3IudHJhbnNhY3Rpb25fZmFtaWxpZXMSngFbeyJmYW1pbHkiOiAiaW50a2V5IiwgIn
header:
  batch_ids:
  -␣
↪a93731646a8fd2bce03b3a17bc2cb3192d8597da93ce735950dccbf0e3cf0b005468fadb94732e013be0bc2afb320b
  block_num: 3
  consensus: RGV2bW9kZQ==
  previous_block_id:␣
↪042f08e1ff49bbf16914a53dc9056fb6e522ca0e2cff872547eac9555c1de2a6200e67fb9daae6dfb90f02bef6a908
  signer_public_key:␣
↪033fbed13b51eafaca8d1a27abc0d4daf14aab8c0cbc1bb4735c01ff80d6581c52
  state_root_hash:␣
↪5d5ea37cbbf8fe793b6ea4c1ba6738f5eee8fc4c73cdca797736f5afeb41fbef
header_signature:␣
↪ff4f6705bf57e2a1498dc1b649cc9b6a4da2cc8367f1b70c02bc6e7f648a28b53b5f6ad7c2aa639673d873959f5d3f
```

Viewing State Data

The `sawtooth state` command lets you display state data. Sawtooth stores state data in a *Merkle-Radix tree*; for more information, see *Global State*.

1. Use `sawtooth state list` to list the nodes (addresses) in state:

   ```
   $ sawtooth state list
   ```

   The output will be similar to this truncated example:

   ```
   ADDRESS                                                                                     ␣
   ↪                                                      SIZE DATA
   1cf126ddb507c936e4ee2ed07aa253c2f4e7487af3a0425f0dc7321f94be02950a081ab7058bf046c788dbaf0f10a980
   ↪11   b'\xa1fcCTdcH\x...
   1cf1260cd1c2492b6e700d5ef65f136051251502e5d4579827dc303f7ed76ddb7185a19be0c6443503594c3734141d2b
   ↪11   b'\xa1frdLONu\x...
   1cf126ed7d0ac4f755be5dd040e2dfcd71c616e697943f542682a2feb14d5f146538c643b19bcfc8c4554c9012e56209
   ↪11   b'\xa1fAUZZqk\x...
   1cf126c46ff13fcd55713bcfcf7b66eba515a51965e9afa8b4ff3743dc6713f4c40b4254df1a2265d64d58afa14a0051
   ↪11   b'\xa1fLvUYLk\x...
   1cf126c4b1b09ebf28775b4923e5273c4c01ba89b961e6a9984632612ec9b5af82a0f7c8fc1a44b9ae33bb88f4ed39b5
   ↪13   b'\xa1fXHonWY\x...
   1cf126e924a506fb2c4bb8d167d20f07d653de2447df2754de9eb61826176c7896205a17e363e457c36ccd2b7c124516
   ↪13   b'\xa1foWZXEz\x...
   1cf126c295a476acf935cd65909ed5ead2ec0168f3ee761dc6f37ea9558fc4e32b71504bf0ad56342a6671db82cb8682
   ↪13   b'\xa1fadKGve\x...
   ```

2. Use `sawtooth state show` to view state data at a specific address (a node in the Merkle-Radix database).

Copy the address from the output of `sawtooth state list`, then paste it in place of `{STATE_ADDRESS}` in the following command:

```
$ sawtooth state show {STATE_ADDRESS}
```

The output shows the bytes stored at that address and the block ID of the "chain head" that the current state is tied to, as in this example:

```
DATA: "b'\xa1fcCTdcH\x192B'"
HEAD:
↪"0c4364c6d5181282a1c7653038ec9515cb0530c6bfcb46f16e79b77cb524491676638339e8ff8e3cc57155c6d920e
↪"
```

Examining Sawtooth Logs

By default, Sawtooth logs are stored in the directory `/var/log/sawtooth`. Each component (validator, REST API, and transaction processors) has both a debug log and an error log. This example shows the log files for the demo environment:

```
$ ls -1 /var/log/sawtooth
identity-f5c42a08548c4ffa-debug.log
identity-f5c42a08548c4ffa-error.log
intkey-ae98c3726f9743c4-debug.log
intkey-ae98c3726f9743c4-error.log
rest_api-debug.log
rest_api-error.log
settings-6d591c44915b465c-debug.log
settings-6d591c44915b465c-error.log
validator-debug.log
validator-error.log
xo-9b8b55265ca0d546-error.log
xo-9b8b55265ca0d546-debug.log
```

Note: The transaction processor log files contain a string to make the file names unique. The file names on your system may be different than these examples.

For more information on log files, see *Log Configuration*.

Resetting the Environment

When you are done with the AWS demo environment (or if you want to reset it), you can use the following commands to restore the Sawtooth instance to its original state.

```
$ sudo rm /var/lib/sawtooth/config-genesis.batch
$ sudo systemctl restart sawtooth-setup.service
```

The first command removes the file `config-genesis.batch`. The second command restarts the `sawtooth-setup` service, which cleans up your validator, creates a new genesis block, and restarts the `sawtooth-validator` service so that you're ready to build on a new blockchain.

3.4 Introduction to the XO Transaction Family

XO is an example transaction family that implements the game tic-tac-toe, also known as *Noughts and Crosses* or *X's and O's*. We chose XO as an example transaction family for Sawtooth because of its simplicity, global player base, and straightforward implementation as a computer program. This transaction family demonstrates the functionality of Sawtooth; in addition, the code that implements it serves as a reference for building other transaction processors.

This section introduces the concepts of a Sawtooth transaction family with XO, summarizes XO game rules, and describes how use the `xo` client application to play a game of tic-tac-toe on the blockchain.

3.4.1 About the XO Transaction Family

The XO transaction family defines the data model and business logic for playing tic-tac-toe on the blockchain by submitting transactions for *create*, *take*, and *delete* actions. For more information, see *XO Transaction Family*

The XO transaction family includes:

- Transaction processors in several languages, including Go (`xo-tp-go`), JavaScript (`xo-tp-js`), and Python (`xo-tp-python`). These transaction processors implement the business logic of XO game play.
- An `xo` client: A set of commands that provide a command-line interface for playing XO. The `xo` client handles the construction and submission of transactions. For more information, see *xo*.

3.4.2 Game Rules

In tic-tac-toe, two players take turns marking spaces on a 3x3 grid.

- The first player (player 1) marks spaces with an X. Player 1 always makes the first move.
- The second player (player 2) marks spaces with an O.
- A player wins the game by marking three adjoining spaces in a horizontal, vertical, or diagonal row.
- The game is a tie if all nine spaces on the grid have been marked, but no player has won.

See Wikipedia for more information on playing tic-tac-toe. For the detailed business logic of game play, see "Execution" in *XO Transaction Family*.

3.4.3 Playing XO with the xo Client

This procedure introduces you to the XO transaction family by playing a game with the `xo` client. Each `xo` command is a transaction that the client submits to the validator via the REST API.

Prerequisites

- A working Sawtooth development environment, as described in *Installing and Running Sawtooth*. Ensure that this environment is running a validator, a REST API, and an XO transaction processor (such as `xo-tp-python`).
- If you are using the Docker development environment described in *Installing and Running Sawtooth*, open a client container by running the following command from your host computer's terminal window:

  ```
  % docker exec -it sawtooth-shell-default bash
  ```

  Otherwise, see your docker-compose file for the correct container name.

- Verify that you can connect to the REST API.
  - Docker: See *Confirming Connectivity*
  - Ubuntu: See *Installing and Running Sawtooth*
  - AWS: See *Checking the Status of Sawtooth Components*

Important: The `xo` client sends requests to update and query the blockchain to the URL of the REST API (by default, `http://127.0.0.1:8008`).

If the REST API's URL is not `http://127.0.0.1:8008`, you must add the `--url` argument to each `xo` command in this procedure.

The following example specifies the URL for the Docker demo application environment when creating a new game:

```
$ xo create my-game --username jack --url http://rest-api:8008
```

Step 1. Create Players

Create keys for two players to play the game:

```
$ sawtooth keygen jack
writing file: /home/ubuntu/.sawtooth/keys/jack.priv
writing file: /home/ubuntu/.sawtooth/keys/jack.addr

$ sawtooth keygen jill
writing file: /home/ubuntu/.sawtooth/keys/jill.priv
writing file: /home/ubuntu/.sawtooth/keys/jill.addr
```

Note: The output may differ slightly from this example.

Step 2. Create a Game

Create a game named `my-game` with the following command:

```
$ xo create my-game --username jack
```

Note: The `--username` argument is required for `xo create` and `xo take` so that a single player (you) can play as two players. By default, `<username>` is the Linux user name of the person playing the game.

Verify that the `create` transaction was committed by displaying the list of existing games:

```
$ xo list
GAME            PLAYER 1        PLAYER 2        BOARD     STATE
my-game                                         --------- P1-NEXT
```

Note: The `xo list` command is a wrapper that provides a quick way to show game state rather than using `curl` with the REST API's URL to request state.

Step 3. Take a Space as Player 1

Note: The first player to issue an `xo take` command to a newly created game is recorded as `PLAYER 1`. The second player to issue a `take` command is recorded by username as `PLAYER 2`.

The `--username` argument determines where the `xo` client should look for the player's key to sign the transaction. By default, if you're logged in as `root`, `xo` would look for the key file named `~/.sawtooth/keys/root.priv`. Instead, the following command specifies that `xo` should use the key file `~/.sawtooth/keys/jack.priv`.

Start playing tic-tac-toe by taking a space as the first player, Jack. In this example, Jack takes space 5:

```
$ xo take my-game 5 --username jack
```

This diagram shows the number of each space.

```
 1 | 2 | 3
---|---|---
 4 | 5 | 6
---|---|---
 7 | 8 | 9
```

What Happens During a Game Move?

Each `xo` command is a transaction. A successful transaction updates global state with the game name, board state, game state, and player keys, using this format:

```
<game-name>,<board-state>,<game-state>,<player1-key>,<player2-key>
```

Each time a player attempts to take a space, the transaction processor will verify that their username matches the name of the player whose turn it is. This ensures that no player is able to mark a space out of turn.

After each turn, the XO transaction processor scans the board state for a win or tie. If either condition occurs, no more `take` actions are allowed on the finished game.

Step 4. Take a Space as Player 2

Next, take a space on the board as player 2, Jill. In this example, Jill takes space 1:

```
$ xo take my-game 1 --username jill
```

Step 5. Show the Current Game Board

Whenever you want to see the current state of the game board, enter the following command:

```
$ xo show my-game
```

The output includes the game name, the first six characters of each player's public key, the game state, and the current board state. This example shows the game state `P1-NEXT` (player 1 has the next turn) and a board with Jack's X in space 5 and Jill's O in space 1.

```
GAME:      : my-game
PLAYER 1   : 02403a
PLAYER 2   : 03729b
```

(continues on next page)

(continued from previous page)

```
STATE     : P1-NEXT

  O |   |
 ---|---|---
    | X |
 ---|---|---
    |   |
```

This xo client formats the global state data so that it's easier to read than the state returned to the transaction processor:

```
my-game,O---X----,P1-NEXT,02403a...,03729b...
```

Step 6. Continue the Game

Players take turns using `xo take my-game <space>` to mark spaces on the grid.

You can continue the game until one of the players wins or the game ends in a tie, as in this example:

```
$ xo show my-game
GAME:     : my-game
PLAYER 1  : 02403a
PLAYER 2  : 03729b
STATE     : TIE

  O | X | O
 ---|---|---
  X | X | O
 ---|---|---
  X | O | X
```

Step 7. Delete the Game

Either player can use the `xo delete` command to remove the game data from global state.

```
$ xo delete my-game
```

3.4.4 Using Authentication with the xo Client

The XO client supports optional authentication. If the REST API is connected to an authentication proxy, you can point the XO client at it with the `--url` argument. You must also specify your authentication information using the `--auth-user [user]` and `--auth-password [password]` options for each `xo` command.

Note that the value of the `--auth-user` argument is **not** the same username that is entered with the `--username` argument.

3.5 Tutorial: Using the Go SDK

This tutorial describes how to develop a Sawtooth application with an example transaction family, using the Sawtooth Go SDK.

A transaction family includes these components:

- A transaction processor to define the business logic for your application. The transaction processor is responsible for registering with the validator, handling transaction payloads and associated metadata, and getting/setting state as needed.
- A data model to record and store data
- A client to handle the client logic for your application. The client is responsible for creating and signing transactions, combining those transactions into batches, and submitting them to the validator. The client can post batches through the REST API or connect directly to the validator via ZeroMQ.

The client and transaction processor must use the same data model, serialization/encoding method, and addressing scheme.

In this tutorial, you will construct a transaction handler that implements XO, a distributed version of the two-player game tic-tac-toe.

Note: This tutorial demonstrates the relevant concepts for a Sawtooth transaction processor and client, but does not create a complete implementation.

- For a full implementation of the tic-tac-toe transaction family, see `/project/sawtooth-core/sdk/examples/xo_go/`.
- For full implementations in other languages, see `https://github.com/hyperledger/sawtooth-core/tree/master/sdk/examples`.

3.5.1 Prerequisites

- A working Sawtooth development environment, as described in *Installing and Running Sawtooth*
- Familiarity with the basic Sawtooth concepts introduced in *Installing and Running Sawtooth*
- Understanding of the Sawtooth transaction and batch data structures as described in *Transactions and Batches*

3.5.2 Transaction Processor: Creating a Transaction Handler

A transaction processor has two top-level components:

- Processor class. The SDK provides a general-purpose processor class.
- Handler class. The handler class is application-dependent. It contains the business logic for a particular family of transactions. Multiple handlers can be connected to an instance of the processor class.

Entry Point

Since a transaction processor is a long running process, it must have an entry point.

In the entry point, the `TransactionProcessor` class is given the address to connect with the validator and the handler class.

Listing 1: a simplified sawtooth_xo/main.go

```
import (
    "sawtooth_sdk/processor"
    xo "sawtooth_xo/handler"
    "syscall"
```

(continues on next page)

(continued from previous page)

```
)

func main() {

    endpoint := "tcp://127.0.0.1:4004"
    // In docker, endpoint would be the validator's container name
    // with port 4004
    handler := &xo.XoHandler{}
    processor := processor.NewTransactionProcessor(endpoint)
    processor.AddHandler(handler)
    processor.ShutdownOnSignal(syscall.SIGINT, syscall.SIGTERM)

    processor.Start()
}
```

Handlers get called in two ways: with an `apply` method and with various “metadata” methods. The metadata is used to connect the handler to the processor. The bulk of the handler, however, is made up of `apply` and its helper functions.

Listing 2: sawtooth_xo/handler/handler.go XoHandler struct

```
type XoHandler struct {
}

func (self *XoHandler) FamilyName() string {
    return "xo"
}

func (self *XoHandler) FamilyVersions() []string {
    return []string{"1.0"}
}

func (self *XoHandler) Namespaces() []string {
    return []string{xo_state.Namespace}
}

func (self *XoHandler) Apply(request *processor_pb2.TpProcessRequest, context␣
↪*processor.Context) error {
```

The `apply` Method

`apply` gets called with two arguments, `request` and `context`. `request` holds the command that is to be executed (e.g. taking a space or creating a game), while `context` stores information about the current state of the game (e.g. the board layout and whose turn it is).

The transaction contains payload bytes that are opaque to the validator core, and transaction family specific. When implementing a transaction handler the binary serialization protocol is up to the implementer.

To separate details of state encoding and payload handling from validation logic, the XO example has `XoState` and `XoPayload` classes. The `XoPayload` has name, action, and space fields, while the `XoState` contains information about the game name, board, state, and which players are playing in the game.

Valid actions are: create a new game, take an unoccupied space, and delete a game.

Listing 3: sawtooth_xo/handler/handler.go apply overview

```
func (self *XoHandler) Apply(request *processor_pb2.TpProcessRequest, context␣
↪*processor.Context) error {
    // The xo player is defined as the signer of the transaction, so we unpack
    // the transaction header to obtain the signer's public key, which will be
    // used as the player's identity.
    header := request.GetHeader()
    player := header.GetSignerPublicKey()

    // The payload is sent to the transaction processor as bytes (just as it
    // appears in the transaction constructed by the transactor).  We unpack
    // the payload into an XoPayload struct so we can access its fields.
    payload, err := xo_payload.FromBytes(request.GetPayload())
    if err != nil {
        return err
    }

    xoState := xo_state.NewXoState(context)

    switch payload.Action {
    case "create":
        ...
    case "delete":
        ...
    case "take":
        ...
    default:
        return &processor.InvalidTransaction{
            Msg: fmt.Sprintf("Invalid Action : '%v'", payload.Action)}
    }
```

For every new payload, the transaction processor validates rules surrounding the action. If all of the rules validate, then state is updated based on whether we are creating a game, deleting a game, or updating the game by taking a space.

Game Logic

The XO game logic is described in the XO transaction family specification; see *Execution*.

The validation rules and state updates that are associated with the `create`, `delete`, and `take` actions are shown below.

Create

The `create` action has the following implementation:

Listing 4: sawtooth_xo/handler/handler.go apply 'create'

```
case "create":
            err := validateCreate(xoState, payload.Name)
            if err != nil {
                    return err
            }
            game := &xo_state.Game{
```

(continues on next page)

(continued from previous page)

```
                    Board:   "---------",
                    State:   "P1-NEXT",
                    Player1: "",
                    Player2: "",
                    Name:    payload.Name,
            }
            displayCreate(payload, player)
    return xoState.SetGame(payload.Name, game)
```

`validateCreate` is defined as follows:

```
func validateCreate(xoState *xo_state.XoState, name string) error {
    game, err := xoState.GetGame(name)
    if err != nil {
        return err
    }
    if game != nil {
        return &processor.InvalidTransactionError{Msg: "Game already exists"}
    }

    return nil
}
```

Delete

The `delete` action has the following implementation:

Listing 5: sawtooth_xo/handler/handler.go apply ‘delete’

```
case "delete":
            err := validateDelete(xoState, payload.Name)
            if err != nil {
                    return err
            }
    return xoState.DeleteGame(payload.Name)
```

`validateDelete` is defined as follows:

```
func validateDelete(xoState *xo_state.XoState, name string) error {
    game, err := xoState.GetGame(name)
    if err != nil {
        return err
    }
    if game == nil {
        return &processor.InvalidTransactionError{Msg: "Delete requires an existing␣
↪game"}
    }
    return nil
}
```

Take

The `take` action has the following implementation:

Listing 6: sawtooth_xo/handler/handler.go apply 'take'

```
case "take":
            err := validateTake(xoState, payload, player)
            if err != nil {
                    return err
            }
            game, err := xoState.GetGame(payload.Name)
            if err != nil {
                    return err
            }
            // Assign players if new game
            if game.Player1 == "" {
                    game.Player1 = player
            } else if game.Player2 == "" {
                    game.Player2 = player
            }

            if game.State == "P1-NEXT" && player == game.Player1 {
                    boardRunes := []rune(game.Board)
                    boardRunes[payload.Space-1] = 'X'
                    game.Board = string(boardRunes)
                    game.State = "P2-NEXT"
            } else if game.State == "P2-NEXT" && player == game.Player2 {
                    boardRunes := []rune(game.Board)
                    boardRunes[payload.Space-1] = 'O'
                    game.Board = string(boardRunes)
                    game.State = "P1-NEXT"
            } else {
                    return &processor.InvalidTransactionError{
                            Msg: fmt.Sprintf("Not this player's turn: '%v'", player)}
            }

            if isWin(game.Board, 'X') {
                    game.State = "P1-WIN"
            } else if isWin(game.Board, 'O') {
                    game.State = "P2-WIN"
            } else if !strings.Contains(game.Board, "-") {
                    game.State = "TIE"
            }
            displayTake(payload, player, game)
    return xoState.SetGame(payload.Name, game)
```

`validateTake` is defined as follows:

```
func validateTake(xoState *xo_state.XoState, payload *xo_payload.XoPayload, signer␣
↪string) error {
    game, err := xoState.GetGame(payload.Name)
    if err != nil {
        return err
    }
    if game == nil {
        return &processor.InvalidTransactionError{Msg: "Take requires an existing game
↪"}
    }
    if game.State == "P1-WIN" || game.State == "P2-WIN" || game.State == "TIE" {
        return &processor.InvalidTransactionError{Msg: "Game has ended"}
```

(continues on next page)

(continued from previous page)

```
    }

    if game.State == "P1-WIN" || game.State == "P2-WIN" || game.State == "TIE" {
        return &processor.InvalidTransactionError{
            Msg: "Invalid Action: Game has ended"}
    }

    if game.Board[payload.Space-1] != '-' {
        return &processor.InvalidTransactionError{Msg: "Space already taken"}
    }
    return nil
}
```

Payload

Note: *Transactions and Batches* contains a detailed description of how transactions are structured and used. Please read this document before proceeding, if you have not reviewed it.

So how do we get data out of the transaction? The transaction consists of a header and a payload. The header contains the “signer”, which is used to identify the current player. The payload will contain an encoding of the game name, the action (`create` a game, `delete` a game, `take` a space), and the space (which will be an empty string if the action isn’t `take`).

An XO transaction request payload consists of the UTF-8 encoding of a string with exactly two commas, formatted as follows:

`<name>,<action>,<space>`

where

- <name> is a nonempty string not containing the character `|`
- <action> is either `take` or `create`
- <space> is an integer strictly between 0 and 10 if the action is `take`

Listing 7: sawtooth_xo/xo_payload/xo_payload.go

```
type XoPayload struct {
    Name   string
    Action string
    Space  int
}

func FromBytes(payloadData []byte) (*XoPayload, error) {
    if payloadData == nil {
        return nil, &processor.InvalidTransactionError{Msg: "Must contain payload"}
    }

    parts := strings.Split(string(payloadData), ",")
    if len(parts) != 3 {
        return nil, &processor.InvalidTransactionError{Msg: "Payload is malformed"}
    }

    payload := XoPayload{}
```

(continues on next page)

(continued from previous page)

```
    payload.Name = parts[0]
    payload.Action = parts[1]

    if len(payload.Name) < 1 {
        return nil, &processor.InvalidTransactionError{Msg: "Name is required"}
    }

    if len(payload.Action) < 1 {
        return nil, &processor.InvalidTransactionError{Msg: "Action is required"}
    }

    if payload.Action == "take" {
        space, err := strconv.Atoi(parts[2])
        if err != nil {
            return nil, &processor.InvalidTransactionError{
                Msg: fmt.Sprintf("Invalid Space: '%v'", parts[2])}
        }
        payload.Space = space
    }

    if strings.Contains(payload.Name, "|") {
        return nil, &processor.InvalidTransactionError{
            Msg: fmt.Sprintf("Invalid Name (char '|' not allowed): '%v'", parts[2])}
    }

    return &payload, nil
}
```

State

The XoState class turns game information into bytes and stores it in the validator's Radix-Merkle tree, turns bytes stored in the validator's Radix-Merkle tree into game information, and does these operations with a state storage scheme that handles hash collisions.

An XO state entry consists of the UTF-8 encoding of a string with exactly four commas formatted as follows:

`<name>,<board>,<game-state>,<player-key-1>,<player-key-2>`

where

- <name> is a nonempty string not containing |,
- <board> is a string of length 9 containing only *O*, *X*, or -,
- <game-state> is one of the following: *P1-NEXT*, *P2-NEXT*, *P1-WIN*,
- *P2-WIN*, or *TIE*, and
- <player-key-1> and <player-key-2> are the (possibly empty) public keys
- associated with the game's players.

In the event of a hash collision (i.e. two or more state entries sharing the same address), the colliding state entries will stored as the UTF-8 encoding of the string `<a-entry>|<b-entry>|...`, where <a-entry>, <b-entry>,… are sorted alphabetically.

Listing 8: sawtooth_xo/xo_state/xo_state.go

```go
var Namespace = hexdigest("xo")[:6]

type Game struct {
    Board   string
    State   string
    Player1 string
    Player2 string
    Name    string
}

// XoState handles addressing, serialization, deserialization,
// and holding an addressCache of data at the address.
type XoState struct {
    context      *processor.Context
    addressCache map[string][]byte
}

// NewXoState constructs a new XoState struct.
func NewXoState(context *processor.Context) *XoState {
    return &XoState{
        context:      context,
        addressCache: make(map[string][]byte),
    }
}

// GetGame returns a game by its name.
func (self *XoState) GetGame(name string) (*Game, error) {
    games, err := self.loadGames(name)
    if err != nil {
        return nil, err
    }
    game, ok := games[name]
    if ok {
        return game, nil
    }
    return nil, nil
}

// SetGame sets a game to its name
func (self *XoState) SetGame(name string, game *Game) error {
    games, err := self.loadGames(name)
    if err != nil {
        return err
    }

    games[name] = game

    return self.storeGames(name, games)
}

// DeleteGame deletes the game from state, handling
// hash collisions.
func (self *XoState) DeleteGame(name string) error {
    games, err := self.loadGames(name)
    if err != nil {
```

(continues on next page)

(continued from previous page)

```
        return err
    }
    delete(games, name)
    if len(games) > 0 {
        return self.storeGames(name, games)
    } else {
        return self.deleteGames(name)
    }
}

func (self *XoState) loadGames(name string) (map[string]*Game, error) {
    address := makeAddress(name)

    data, ok := self.addressCache[address]
    if ok {
        if self.addressCache[address] != nil {
            return deserialize(data)
        }
        return make(map[string]*Game), nil

    }
    results, err := self.context.GetState([]string{address})
    if err != nil {
        return nil, err
    }
    if len(string(results[address])) > 0 {
        self.addressCache[address] = results[address]
        return deserialize(results[address])
    }
    self.addressCache[address] = nil
    games := make(map[string]*Game)
    return games, nil
}

func (self *XoState) storeGames(name string, games map[string]*Game) error {
    address := makeAddress(name)

    var names []string
    for name := range games {
        names = append(names, name)
    }
    sort.Strings(names)

    var g []*Game
    for _, name := range names {
        g = append(g, games[name])
    }

    data := serialize(g)

    self.addressCache[address] = data

    _, err := self.context.SetState(map[string][]byte{
        address: data,
    })
    return err
}
```

(continues on next page)

(continued from previous page)

```
func (self *XoState) deleteGames(name string) error {
    address := makeAddress(name)

    _, err := self.context.DeleteState([]string{address})
    return err
}

func deserialize(data []byte) (map[string]*Game, error) {
    games := make(map[string]*Game)
    for _, str := range strings.Split(string(data), "|") {

        parts := strings.Split(string(str), ",")
        if len(parts) != 5 {
            return nil, &processor.InternalError{
                Msg: fmt.Sprintf("Malformed game data: '%v'", string(data))}
        }

        game := &Game{
            Name:    parts[0],
            Board:   parts[1],
            State:   parts[2],
            Player1: parts[3],
            Player2: parts[4],
        }
        games[parts[0]] = game
    }

    return games, nil
}

func serialize(games []*Game) []byte {
    var buffer bytes.Buffer
    for i, game := range games {

        buffer.WriteString(game.Name)
        buffer.WriteString(",")
        buffer.WriteString(game.Board)
        buffer.WriteString(",")
        buffer.WriteString(game.State)
        buffer.WriteString(",")
        buffer.WriteString(game.Player1)
        buffer.WriteString(",")
        buffer.WriteString(game.Player2)
        if i+1 != len(games) {
            buffer.WriteString("|")
        }
    }
    return buffer.Bytes()
}

func hexdigest(str string) string {
    hash := sha512.New()
    hash.Write([]byte(str))
    hashBytes := hash.Sum(nil)
    return strings.ToLower(hex.EncodeToString(hashBytes))
}
```

Addressing

By convention, we'll store game data at an address obtained from hashing the game name prepended with some constant.

XO data is stored in state using addresses generated from the XO family name and the name of the game being stored. In particular, an XO address consists of the first 6 characters of the SHA-512 hash of the UTF-8 encoding of the string "xo" (which is "5b7349") plus the first 64 characters of the SHA-512 hash of the UTF-8 encoding of the game name.

For example, the XO address for a game called "my-game" could be generated as follows:

```
>>> x = hashlib.sha512('xo'.encode('utf-8')).hexdigest()[:6]
>>> x
'5b7349'
>>> y = hashlib.sha512('my-game'.encode('utf-8')).hexdigest()[:64]
>>> y
'4d4cffe9cf3fb4e41def5114a323e292af9b0e07925cca6299d671ce7fc7ec37'
>>> x+y
'5b73494d4cffe9cf3fb4e41def5114a323e292af9b0e07925cca6299d671ce7fc7ec37'
```

Addressing is implemented as follows:

```
func makeAddress(name string) string {
        return Namespace + hexdigest(name)[:64]
}
```

3.6 Tutorial: Using the JavaScript SDK

This tutorial describes how to develop a Sawtooth application with an example transaction family, using the Sawtooth JavaScript SDK.

A transaction family includes these components:

- A transaction processor to define the business logic for your application. The transaction processor is responsible for registering with the validator, handling transaction payloads and associated metadata, and getting/setting state as needed.
- A data model to record and store data
- A client to handle the client logic for your application. The client is responsible for creating and signing transactions, combining those transactions into batches, and submitting them to the validator. The client can post batches through the REST API or connect directly to the validator via ZeroMQ.

The client and transaction processor must use the same data model, serialization/encoding method, and addressing scheme.

In this tutorial, you will construct a transaction handler that implements XO, a distributed version of the two-player game tic-tac-toe.

This tutorial also describes how a client can use the JavaScript SDK to create transactions and submit them as *Sawtooth batches*.

Note: This tutorial demonstrates the relevant concepts for a Sawtooth transaction processor and client, but does not create a complete implementation.

- For a full implementation of the tic-tac-toe transaction family, see `/project/sawtooth-core/sdk/examples/xo_javascript/`.

- For full implementations in other languages, see `https://github.com/hyperledger/sawtooth-core/tree/master/sdk/examples`.

3.6.1 Prerequisites

- A working Sawtooth development environment, as described in *Installing and Running Sawtooth*
- Familiarity with the basic Sawtooth concepts introduced in *Installing and Running Sawtooth*
- Understanding of the Sawtooth transaction and batch data structures as described in *Transactions and Batches*

3.6.2 Transaction Processor: Creating a Transaction Handler

A transaction processor has two top-level components:

- Processor class. The SDK provides a general-purpose processor class.
- Handler class. The handler class is application-dependent. It contains the business logic for a particular family of transactions. Multiple handlers can be connected to an instance of the processor class.

Entry Point

Since a transaction processor is a long running process, it must have an entry point.

In the entry point, the `TransactionProcessor` class is given the address to connect with the validator and the handler class.

Listing 9: a simplified xo_javascript/index.js

```
const { TransactionProcessor } = require('sawtooth-sdk/processor')
const XOHandler = require('./xo_handler')

// In docker, the address would be the validator's container name
// with port 4004
const address = 'tcp://127.0.0.1:4004'
const transactionProcessor = new TransactionProcessor(address)

transactionProcessor.addHandler(new XOHandler())

transactionProcessor.start()
```

Handlers get called in two ways: with an `apply` method and with various "metadata" methods. The metadata is used to connect the handler to the processor. The bulk of the handler, however, is made up of `apply` and its helper functions.

Listing 10: xo_javascript/xo_handler.js XOHandler class

```
class XOHandler extends TransactionHandler {
  constructor () {
    super(XO_FAMILY, '1.0', 'csv-utf8', [XO_NAMESPACE])
  }

  apply (transactionProcessRequest, stateStore) {
    //
```

Note that the XOHandler class extends the TransactionHandler class defined in the JavaScript SDK.

The apply Method

apply gets called with two arguments, transactionProcessRequest and stateStore. transactionProcessRequest holds the command that is to be executed (e.g. taking a space or creating a game), while stateStore stores information about the current state of the game (e.g. the board layout and whose turn it is).

The transaction contains payload bytes that are opaque to the validator core, and transaction family specific. When implementing a transaction handler the binary serialization protocol is up to the implementer.

To separate details of state encoding and payload handling from validation logic, the XO example has XoState and XoPayload classes. The XoPayload has name, action, and space fields, while the XoState contains information about the game name, board, state, and which players are playing in the game.

Valid actions are: create a new game, take an unoccupied space, and delete a game.

Listing 11: xo_javascript/xo_handler.js apply overview

```
apply (transactionProcessRequest, context) {
    let payload = XoPayload.fromBytes(transactionProcessRequest.payload)
    let xoState = new XoState(context)
    let header = transactionProcessRequest.header
    let player = header.signerPublicKey
    if (payload.action === 'create') {
        ...
    } else if (payload.action === 'take') {
        ...
    } else if (payload.action === 'delete') {
        ...
    } else {
        throw new InvalidTransaction(
            `Action must be create, delete, or take not ${payload.action}`
        )
    }
}
```

For every new payload, the transaction processor validates rules surrounding the action. If all of the rules validate, then state is updated based on whether we are creating a game, deleting a game, or updating the game by taking a space.

Game Logic

The XO game logic is described in the XO transaction family specification; see *Execution*.

The validation rules and state updates that are associated with the create, delete, and take actions are shown below.

Create

The create action has the following implementation:

Listing 12: xo_javascript/xo_handler.js apply 'create'

```
if (payload.action === 'create') {
  return xoState.getGame(payload.name)
    .then((game) => {
      if (game !== undefined) {
        throw new InvalidTransaction('Invalid Action: Game already exists.')
      }

      let createdGame = {
        name: payload.name,
        board: '---------',
        state: 'P1-NEXT',
        player1: '',
        player2: ''
      }

      _display(`Player ${player.toString().substring(0, 6)} created game ${payload.
↪name}`)

      return xoState.setGame(payload.name, createdGame)
    })
}
```

Delete

The `delete` action has the following implementation:

Listing 13: xo_javascript/xo_handler.js apply 'delete'

```
if (payload.action === 'delete') {
  return xoState.getGame(payload.name)
    .then((game) => {
      if (game === undefined) {
        throw new InvalidTransaction(
          `No game exists with name ${payload.name}: unable to delete`)
      }
      return xoState.deleteGame(payload.name)
    })
} else {
  throw new InvalidTransaction(
    `Action must be create or take not ${payload.action}`
  )
}
```

Take

The `take` action has the following implementation:

Listing 14: xo_javascript/xo_handler.js apply 'take'

```
if (payload.action === 'take') {
  return xoState.getGame(payload.name)
    .then((game) => {
```

(continues on next page)

(continued from previous page)

```
      try {
        parseInt(payload.space)
      } catch (err) {
        throw new InvalidTransaction('Space could not be converted as an integer.')
      }

      if (payload.space < 1 || payload.space > 9) {
        throw new InvalidTransaction('Invalid space ' + payload.space)
      }

      if (game === undefined) {
        throw new InvalidTransaction(
          'Invalid Action: Take requires an existing game.'
        )
      }
      if (['P1-WIN', 'P2-WIN', 'TIE'].includes(game.state)) {
        throw new InvalidTransaction('Invalid Action: Game has ended.')
      }

      if (game.player1 === '') {
        game.player1 = player
      } else if (game.player2 === '') {
        game.player2 = player
      }
      let boardList = game.board.split('')

      if (boardList[payload.space - 1] !== '-') {
        throw new InvalidTransaction('Invalid Action: Space already taken.')
      }

      if (game.state === 'P1-NEXT' && player === game.player1) {
        boardList[payload.space - 1] = 'X'
        game.state = 'P2-NEXT'
      } else if (
        game.state === 'P2-NEXT' &&
        player === game.player2
      ) {
        boardList[payload.space - 1] = 'O'
        game.state = 'P1-NEXT'
      } else {
        throw new InvalidTransaction(
          `Not this player's turn: ${player.toString().substring(0, 6)}`
        )
      }

      game.board = boardList.join('')

      if (_isWin(game.board, 'X')) {
        game.state = 'P1-WIN'
      } else if (_isWin(game.board, 'O')) {
        game.state = 'P2-WIN'
      } else if (game.board.search('-') === -1) {
        game.state = 'TIE'
      }

      let playerString = player.toString().substring(0, 6)
```

(continues on next page)

(continued from previous page)

```
      _display(
        `Player ${playerString} takes space: ${payload.space}\n\n` +
          _gameToStr(
            game.board,
            game.state,
            game.player1,
            game.player2,
            payload.name
          )
      )

      return xoState.setGame(payload.name, game)
    })
}
```

Payload

Note: *Transactions and Batches* contains a detailed description of how transactions are structured and used. Please read this document before proceeding, if you have not reviewed it.

So how do we get data out of the transaction? The transaction consists of a header and a payload. The header contains the "signer", which is used to identify the current player. The payload will contain an encoding of the game name, the action (`create` a game, `delete` a game, `take` a space), and the space (which will be an empty string if the action isn't `take`).

An XO transaction request payload consists of the UTF-8 encoding of a string with exactly two commas, formatted as follows:

`<name>,<action>,<space>`

where

- <name> is a nonempty string not containing the character `|`
- <action> is either `take` or `create`
- <space> is an integer strictly between 0 and 10 if the action is `take`

Listing 15: xo_javascript/xo_payload.js

```
class XoPayload {
    constructor (name, action, space) {
        this.name = name
        this.action = action
        this.space = space
    }

    static fromBytes (payload) {
        payload = payload.toString().split(',')
        if (payload.length === 3) {
            let xoPayload = new XoPayload(payload[0], payload[1], payload[2])
            if (!xoPayload.name) {
                throw new InvalidTransaction('Name is required')
            }
            if (xoPayload.name.indexOf('|') !== -1) {
```

(continues on next page)

(continued from previous page)

```
                throw new InvalidTransaction('Name cannot contain "|"')
            }

            if (!xoPayload.action) {
                throw new InvalidTransaction('Action is required')
            }
            return xoPayload
        } else {
        throw new InvalidTransaction('Invalid payload serialization')
        }
    }
}
```

State

The XoState class turns game information into bytes and stores it in the validator's Radix-Merkle tree, turns bytes stored in the validator's Radix-Merkle tree into game information, and does these operations with a state storage scheme that handles hash collisions.

An XO state entry consists of the UTF-8 encoding of a string with exactly four commas formatted as follows:

`<name>,<board>,<game-state>,<player-key-1>,<player-key-2>`

where

- <name> is a nonempty string not containing |,
- <board> is a string of length 9 containing only *O*, *X*, or -,
- <game-state> is one of the following: *P1-NEXT*, *P2-NEXT*, *P1-WIN*,
- *P2-WIN*, or *TIE*, and
- <player-key-1> and <player-key-2> are the (possibly empty) public keys
- associated with the game's players.

In the event of a hash collision (i.e. two or more state entries sharing the same address), the colliding state entries will stored as the UTF-8 encoding of the string `<a-entry>|<b-entry>|...`, where <a-entry>, <b-entry>,… are sorted alphabetically.

Listing 16: xo_javascript/xo_state.js

```
class XoState {
    constructor (context) {
        this.context = context
        this.addressCache = new Map([])
        this.timeout = 500 // Timeout in milliseconds
    }

    getGame (name) {
        return this._loadGames(name).then((games) => games.get(name))
    }

    setGame (name, game) {
        let address = _makeXoAddress(name)

        return this._loadGames(name).then((games) => {
```

(continues on next page)

(continued from previous page)

```
            games.set(name, game)
            return games
        }).then((games) => {
            let data = _serialize(games)

            this.addressCache.set(address, data)
            let entries = {
                [address]: data
            }
            return this.context.setState(entries, this.timeout)
        })
    }

    deleteGame (name) {
        let address = _makeXoAddress(name)
        return this._loadGames(name).then((games) => {
            games.delete(name)

            if (games.size === 0) {
                this.addressCache.set(address, null)
                return this.context.deleteState([address], this.timeout)
            } else {
                let data = _serialize(games)
                this.addressCache.set(address, data)
                let entries = {
                    [address]: data
                }
                    return this.context.setState(entries, this.timeout)
                }
            })
        }

    _loadGames (name) {
        let address = _makeXoAddress(name)
        if (this.addressCache.has(address)) {
            if (this.addressCache.get(address) === null) {
                return Promise.resolve(new Map([]))
            } else {
                return Promise.resolve(_deserialize(this.addressCache.get(address)))
            }
        } else {
            return this.context.getState([address], this.timeout)
                .then((addressValues) => {
                    if (!addressValues[address].toString()) {
                        this.addressCache.set(address, null)
                        return new Map([])
                    } else {
                        let data = addressValues[address].toString()
                        this.addressCache.set(address, data)
                        return _deserialize(data)
                    }
                })
            }
        }
    }

const _hash = (x) =>
```

(continues on next page)

(continued from previous page)

```
    crypto.createHash('sha512').update(x).digest('hex').toLowerCase().substring(0, 64)

const XO_FAMILY = 'xo'

const XO_NAMESPACE = _hash(XO_FAMILY).substring(0, 6)

const _deserialize = (data) => {
    let gamesIterable = data.split('|').map(x => x.split(','))
        .map(x => [x[0], {name: x[0], board: x[1], state: x[2], player1: x[3],␣
↪player2: x[4]}])
    return new Map(gamesIterable)
}

const _serialize = (games) => {
    let gameStrs = []
    for (let nameGame of games) {
        let name = nameGame[0]
        let game = nameGame[1]
        gameStrs.push([name, game.board, game.state, game.player1, game.player2].join(
↪','))
    }

    gameStrs.sort()

    return Buffer.from(gameStrs.join('|'))
}
```

Addressing

By convention, we'll store game data at an address obtained from hashing the game name prepended with some constant.

XO data is stored in state using addresses generated from the XO family name and the name of the game being stored. In particular, an XO address consists of the first 6 characters of the SHA-512 hash of the UTF-8 encoding of the string "xo" (which is "5b7349") plus the first 64 characters of the SHA-512 hash of the UTF-8 encoding of the game name.

For example, the XO address for a game called "my-game" could be generated as follows:

```
>>> x = hashlib.sha512('xo'.encode('utf-8')).hexdigest()[:6]
>>> x
'5b7349'
>>> y = hashlib.sha512('my-game'.encode('utf-8')).hexdigest()[:64]
>>> y
'4d4cffe9cf3fb4e41def5114a323e292af9b0e07925cca6299d671ce7fc7ec37'
>>> x+y
'5b73494d4cffe9cf3fb4e41def5114a323e292af9b0e07925cca6299d671ce7fc7ec37'
```

Addressing is implemented as follows:

```
const _makeXoAddress = (x) => XO_NAMESPACE + _hash(x)
```

3.6.3 Client: Building and Submitting Transactions

The process of encoding information to be submitted to a distributed ledger is generally non-trivial. A series of cryptographic safeguards are used to confirm identity and data validity. Hyperledger Sawtooth is no different, but the JavaScript SDK does provide client functionality that abstracts away most of these details, and greatly simplifies the process of making changes to the blockchain.

Creating a Private Key and Signer

In order to confirm your identity and sign the information you send to the validator, you will need a 256-bit key. Sawtooth uses the secp256k1 ECDSA standard for signing, which means that almost any set of 32 bytes is a valid key. It is fairly simple to generate a valid key using the SDK's *signing* module.

A *Signer* wraps a private key and provides some convenient methods for signing bytes and getting the private key's associated public key.

```
const {createContext, CryptoFactory} = require('sawtooth-sdk/signing')

const context = createContext('secp256k1')
const privateKey = context.newRandomPrivateKey()
const signer = CryptoFactory(context).newSigner(privateKey)
```

Note: This key is the **only** way to prove your identity on the blockchain. Any person possessing it will be able to sign Transactions using your identity, and there is no way to recover it if lost. It is very important that any private key is kept secret and secure.

Encoding Your Payload

Transaction payloads are composed of binary-encoded data that is opaque to the validator. The logic for encoding and decoding them rests entirely within the particular Transaction Processor itself. As a result, there are many possible formats, and you will have to look to the definition of the Transaction Processor itself for that information. As an example, the *IntegerKey* Transaction Processor uses a payload of three key/value pairs encoded as CBOR. Creating one might look like this:

```
const cbor = require('cbor')

const payload = {
    Verb: 'set',
    Name: 'foo',
    Value: 42
}

const payloadBytes = cbor.encode(payload)
```

Building the Transaction

Transactions are the basis for individual changes of state to the Sawtooth blockchain. They are composed of a binary payload, a binary-encoded *TransactionHeader* with some cryptographic safeguards and metadata about how it should be handled, and a signature of that header. It would be worthwhile to familiarize yourself with the information in *Transactions and Batches*, particularly the definition of TransactionHeaders.

1. Create the Transaction Header

A TransactionHeader contains information for routing a transaction to the correct transaction processor, what input and output state addresses are involved, references to prior transactions it depends on, and the public keys associated with the its signature. The header references the payload through a SHA-512 hash of the payload bytes.

```
const {createHash} = require('crypto')
const {protobuf} = require('sawtooth-sdk')

const transactionHeaderBytes = protobuf.TransactionHeader.encode({
    familyName: 'intkey',
    familyVersion: '1.0',
    inputs: ['1cf1266e282c41be5e4254d8820772c5518a2c5a8c0c7f7eda19594a7eb539453e1ed7
↪'],
    outputs: ['1cf1266e282c41be5e4254d8820772c5518a2c5a8c0c7f7eda19594a7eb539453e1ed7
↪'],
    signerPublicKey: signer.getPublicKey().asHex(),
    // In this example, we're signing the batch with the same private key,
    // but the batch can be signed by another party, in which case, the
    // public key will need to be associated with that key.
    batcherPublicKey: signer.getPublicKey().asHex(),
    // In this example, there are no dependencies.  This list should include
    // an previous transaction header signatures that must be applied for
    // this transaction to successfully commit.
    // For example,
    // dependencies: [
↪'540a6803971d1880ec73a96cb97815a95d374cbad5d865925e5aa0432fcf1931539afe10310c122c5eaae15df61236079
↪'],
    dependencies: [],
    payloadSha512: createHash('sha512').update(payloadBytes).digest('hex')
}).finish()
```

Note: Remember that a *batcher public_key* is the hex public key matching the private key that will later be used to sign a Transaction's Batch, and *dependencies* are the *header signatures* of Transactions that must be committed before this one (see *TransactionHeaders* in *Transactions and Batches*).

Note: The *inputs* and *outputs* are the state addresses a Transaction is allowed to read from or write to. With the Transaction above, we referenced the specific address where the value of `'foo'` is stored. Whenever possible, specific addresses should be used, as this will allow the validator to schedule transaction processing more efficiently.

Note that the methods for assigning and validating addresses are entirely up to the Transaction Processor. In the case of IntegerKey, there are specific rules to generate valid addresses, which must be followed or Transactions will be rejected. You will need to follow the addressing rules for whichever Transaction Family you are working with.

2. Create the Transaction

Once the TransactionHeader is constructed, its bytes are then used to create a signature. This header signature also acts as the ID of the transaction. The header bytes, the header signature, and the payload bytes are all used to construct the complete Transaction.

```
const signature = signer.sign(transactionHeaderBytes)

const transaction = protobuf.Transaction.create({
    header: transactionHeaderBytes,
    headerSignature: signature,
    payload: payloadBytes
})
```

3. (optional) Encode the Transaction(s)

If the same machine is creating Transactions and Batches there is no need to encode the Transaction instances. However, in the use case where Transactions are being batched externally, they must be serialized before being transmitted to the batcher. The JavaScript SDK offers two options for this. One or more Transactions can be combined into a serialized *TransactionList* method, or can be serialized as a single Transaction.

```
const txnListBytes = protobuf.TransactionList.encode([
    transaction1,
    transaction2
]).finish()

const txnBytes2 = transaction.finish()
```

Building the Batch

Once you have one or more Transaction instances ready, they must be wrapped in a *Batch*. Batches are the atomic unit of change in Sawtooth's state. When a Batch is submitted to a validator each Transaction in it will be applied (in order), or *no* Transactions will be applied. Even if your Transactions are not dependent on any others, they cannot be submitted directly to the validator. They must all be wrapped in a Batch.

1. Create the BatchHeader

Similar to the TransactionHeader, there is a *BatchHeader* for each Batch. As Batches are much simpler than Transactions, a BatchHeader needs only the public key of the signer and the list of Transaction IDs, in the same order they are listed in the Batch.

```
const transactions = [transaction]

const batchHeaderBytes = protobuf.BatchHeader.encode({
    signerPublicKey: signer.getPublicKey().asHex(),
    transactionIds: transactions.map((txn) => txn.headerSignature),
}).finish()
```

2. Create the Batch

Using the SDK, creating a Batch is similar to creating a transaction. The header is signed, and the resulting signature acts as the Batch's ID. The Batch is then constructed out of the header bytes, the header signature, and the transactions that make up the batch.

```
const signature = signer.sign(batchHeaderBytes)

const batch = protobuf.Batch.create({
    header: batchHeaderBytes,
    headerSignature: signature,
    transactions: transactions
})
```

3. Encode the Batch(es) in a BatchList

In order to submit Batches to the validator, they must be collected into a *BatchList*. Multiple batches can be submitted in one BatchList, though the Batches themselves don't necessarily need to depend on each other. Unlike Batches, a BatchList is not atomic. Batches from other clients may be interleaved with yours.

```
const batchListBytes = protobuf.BatchList.encode({
    batches: [batch]
}).finish()
```

Note: Note, if the transaction creator is using a different private key than the batcher, the *batcher public_key* must have been specified for every Transaction, and must have been generated from the private key being used to sign the Batch, or validation will fail.

Submitting Batches to the Validator

The prescribed way to submit Batches to the validator is via the REST API. This is an independent process that runs alongside a validator, allowing clients to communicate using HTTP/JSON standards. Simply send a *POST* request to the */batches* endpoint, with a *"Content-Type"* header of *"application/octet-stream"*, and the *body* as a serialized *BatchList*.

There are a many ways to make an HTTP request, and hopefully the submission process is fairly straightforward from here, but as an example, this is what it might look if you sent the request from the same JavaScript process that prepared the BatchList:

```
const request = require('request')

request.post({
    url: 'http://rest.api.domain/batches',
    body: batchListBytes,
    headers: {'Content-Type': 'application/octet-stream'}
}, (err, response) => {
    if (err) return console.log(err)
    console.log(response.body)
})
```

And here is what it would look like if you saved the binary to a file, and then sent it from the command line with `curl`:

```
const fs = require('fs')

const fileStream = fs.createWriteStream('intkey.batches')
fileStream.write(batchListBytes)
fileStream.end()
```

```
% curl --request POST \
    --header "Content-Type: application/octet-stream" \
    --data-binary @intkey.batches \
    "http://rest.api.domain/batches"
```

3.7 Tutorial: Using the Python SDK

This tutorial describes how to develop a Sawtooth application with an example transaction family, using the Sawtooth Python SDK.

A transaction family includes these components:

- A transaction processor to define the business logic for your application. The transaction processor is responsible for registering with the validator, handling transaction payloads and associated metadata, and getting/setting state as needed.
- A data model to record and store data
- A client to handle the client logic for your application. The client is responsible for creating and signing transactions, combining those transactions into batches, and submitting them to the validator. The client can post batches through the REST API or connect directly to the validator via ZeroMQ.

The client and transaction processor must use the same data model, serialization/encoding method, and addressing scheme.

In this tutorial, you will construct a transaction handler that implements XO, a distributed version of the two-player game tic-tac-toe.

This tutorial also describes how a client can use the Python SDK to create transactions and submit them as *Sawtooth batches*.

Note: This tutorial demonstrates the relevant concepts for a Sawtooth transaction processor and client, but does not create a complete implementation.

- For a full implementation of the tic-tac-toe transaction family, see `/project/sawtooth-core/sdk/examples/xo_python/`.
- For full implementations in other languages, see `https://github.com/hyperledger/sawtooth-core/tree/master/sdk/examples`.

3.7.1 Prerequisites

- A working Sawtooth development environment, as described in *Installing and Running Sawtooth*
- Familiarity with the basic Sawtooth concepts introduced in *Installing and Running Sawtooth*
- Understanding of the Sawtooth transaction and batch data structures as described in *Transactions and Batches*

3.7.2 Importing the Python SDK

Note: The Sawtooth Python SDK requires Python version 3.5 or higher

The Python SDK is installed automatically in the demo development environment, as described by *Installing and Running Sawtooth*. This SDK is available through the standard Python import system.

You can use the Python REPL to import the SDK into your Python environment, then verify the import by viewing the SDK's docstring.

```
$ python3
>>> import sawtooth_sdk
>>> help(sawtooth_sdk)
Help on package sawtooth_sdk:

NAME
    sawtooth_sdk

DESCRIPTION
    # Copyright 2016 Intel Corporation
    #
    # Licensed under the Apache License, Version 2.0 (the "License");
    # you may not use this file except in compliance with the License.
    # You may obtain a copy of the License at
    #
    #     http://www.apache.org/licenses/LICENSE-2.0
    #
    # Unless required by applicable law or agreed to in writing, software
    # distributed under the License is distributed on an "AS IS" BASIS,
    # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    # See the License for the specific language governing permissions and
    # limitations under the License.
    # ------------------------------------------------------------------------------

PACKAGE CONTENTS
    client (package)
    processor (package)
    protobuf (package)
    workload (package)

DATA
    __all__ = ['client', 'processor']

FILE
    /usr/lib/python3/dist-packages/sawtooth_sdk/__init__.py
```

3.7.3 Transaction Processor: Creating a Transaction Handler

A transaction processor has two top-level components:

- Processor class. The SDK provides a general-purpose processor class.
- Handler class. The handler class is application-dependent. It contains the business logic for a particular family of transactions. Multiple handlers can be connected to an instance of the processor class.

Entry Point

Since a transaction processor is a long running process, it must have an entry point.

In the entry point, the `TransactionProcessor` class is given the address to connect with the validator and the handler class.

Listing 17: a simplified sawtooth_xo/processor/main.py

```
from sawtooth_sdk.processor.core import TransactionProcessor
from sawtooth_xo.processor.handler import XoTransactionHandler

def main():
    # In docker, the url would be the validator's container name with
    # port 4004
    processor = TransactionProcessor(url='tcp://127.0.0.1:4004')

    handler = XoTransactionHandler()

    processor.add_handler(handler)

    processor.start()
```

Handlers get called in two ways: with an `apply` method and with various "metadata" methods. The metadata is used to connect the handler to the processor. The bulk of the handler, however, is made up of `apply` and its helper functions.

Listing 18: sawtooth_xo/processor/handler.py XoTransactionHandler class

```
class XoTransactionHandler(TransactionHandler):
    def __init__(self, namespace_prefix):
        self._namespace_prefix = namespace_prefix

    @property
    def family_name(self):
        return 'xo'

    @property
    def family_versions(self):
        return ['1.0']

    @property
    def encodings(self):
        return ['csv-utf8']

    @property
    def namespaces(self):
        return [self._namespace_prefix]

    def apply(self, transaction, context):
        # ...
```

Note that the `XoTransactionHandler` extends the `TransactionHandler` defined in the Python SDK.

The `apply` Method

`apply` gets called with two arguments, `transaction` and `context`. The argument `transaction` is an instance of the class Transaction that is created from the protobuf definition. Also, `context` is an instance of the class Context from the python SDK.

`transaction` holds the command that is to be executed (e.g. taking a space or creating a game), while `context` stores information about the current state of the game (e.g. the board layout and whose turn it is).

The transaction contains payload bytes that are opaque to the validator core, and transaction family specific. When implementing a transaction handler the binary serialization protocol is up to the implementer.

To separate details of state encoding and payload handling from validation logic, the XO example has `XoState` and `XoPayload` classes. The `XoPayload` has name, action, and space fields, while the `XoState` contains information about the game name, board, state, and which players are playing in the game.

Valid actions are: create a new game, take an unoccupied space, and delete a game.

Listing 19: sawtooth_xo/processor/handler.py apply overview

```
def apply(self, transaction, context):

    header = transaction.header
    signer = header.signer_public_key

    xo_payload = XoPayload.from_bytes(transaction.payload)

    xo_state = XoState(context)

    if xo_payload.action == 'delete':
        ...
    elif xo.payload.action == 'create':
        ...
    elif xo.payload.action == 'take':
        ...
    else:
        raise InvalidTransaction('Unhandled action: {}'.format(
            xo_payload.action))
```

For every new payload, the transaction processor validates rules surrounding the action. If all of the rules validate, then state is updated based on whether we are creating a game, deleting a game, or updating the game by taking a space.

Game Logic

The XO game logic is described in the XO transaction family specification; see *Execution*.

The validation rules and state updates that are associated with the `create`, `delete`, and `take` actions are shown below.

Create

The `create` action has the following implementation:

Listing 20: sawtooth_xo/processor/handler.py apply 'create'

```
elif xo_payload.action == 'create':

    if xo_state.get_game(xo_payload.name) is not None:
        raise InvalidTransaction(
            'Invalid action: Game already exists: {}'.format(
                xo_payload.name))

    game = Game(name=xo_payload.name,
                board="-" * 9,
```

(continues on next page)

(continued from previous page)

```
                state="P1-NEXT",
                player1="",
                player2="")

    xo_state.set_game(xo_payload.name, game)
    _display("Player {} created a game.".format(signer[:6]))
```

Delete

The `delete` action has the following implementation:

Listing 21: sawtooth_xo/processor/handler.py apply 'delete'

```
if xo_payload.action == 'delete':
    game = xo_state.get_game(xo_payload.name)

    if game is None:
        raise InvalidTransaction(
            'Invalid action: game does not exist')

    xo_state.delete_game(xo_payload.name)
```

Take

The `take` action has the following implementation:

Listing 22: sawtooth_xo/processor/handler.py apply 'take'

```
elif xo_payload.action == 'take':
    game = xo_state.get_game(xo_payload.name)

    if game is None:
        raise InvalidTransaction(
            'Invalid action: Take requires an existing game')

    if game.state in ('P1-WIN', 'P2-WIN', 'TIE'):
        raise InvalidTransaction('Invalid Action: Game has ended')

    if (game.player1 and game.state == 'P1-NEXT' and
        game.player1 != signer) or \
            (game.player2 and game.state == 'P2-NEXT' and
                game.player2 != signer):
        raise InvalidTransaction(
            "Not this player's turn: {}".format(signer[:6]))

    if game.board[xo_payload.space - 1] != '-':
        raise InvalidTransaction(
            'Invalid Action: space {} already taken'.format(
                xo_payload))

    if game.player1 == '':
        game.player1 = signer
```

(continues on next page)

(continued from previous page)

```
    elif game.player2 == '':
        game.player2 = signer

    upd_board = _update_board(game.board,
                                xo_payload.space,
                                game.state)

    upd_game_state = _update_game_state(game.state, upd_board)

    game.board = upd_board
    game.state = upd_game_state

    xo_state.set_game(xo_payload.name, game)
    _display(
        "Player {} takes space: {}\n\n".format(
            signer[:6],
            xo_payload.space) +
        _game_data_to_str(
            game.board,
            game.state,
            game.player1,
            game.player2,
            xo_payload.name))
```

Payload

Note: *Transactions and Batches* contains a detailed description of how transactions are structured and used. Please read this document before proceeding, if you have not reviewed it.

So how do we get data out of the transaction? The transaction consists of a header and a payload. The header contains the "signer", which is used to identify the current player. The payload will contain an encoding of the game name, the action (`create` a game, `delete` a game, `take` a space), and the space (which will be an empty string if the action isn't `take`).

An XO transaction request payload consists of the UTF-8 encoding of a string with exactly two commas, formatted as follows:

`<name>,<action>,<space>`

where

- <name> is a nonempty string not containing the character `|`
- <action> is either `take` or `create`
- <space> is an integer strictly between 0 and 10 if the action is `take`

Listing 23: sawtooth_xo/processor/xo_payload.py

```
class XoPayload(object):

    def __init__(self, payload):
        try:
            # The payload is csv utf-8 encoded string
            name, action, space = payload.decode().split(",")
```

(continues on next page)

(continued from previous page)

```
        except ValueError:
            raise InvalidTransaction("Invalid payload serialization")

        if not name:
            raise InvalidTransaction('Name is required')

        if '|' in name:
            raise InvalidTransaction('Name cannot contain "|"')

        if not action:
            raise InvalidTransaction('Action is required')

        if action not in ('create', 'take', 'delete'):
            raise InvalidTransaction('Invalid action: {}'.format(action))

        if action == 'take':
            try:

                if int(space) not in range(1, 10):
                    raise InvalidTransaction(
                        "Space must be an integer from 1 to 9")
            except ValueError:
                raise InvalidTransaction(
                    'Space must be an integer from 1 to 9')

        if action == 'take':
            space = int(space)

        self._name = name
        self._action = action
        self._space = space

    @staticmethod
    def from_bytes(payload):
        return XoPayload(payload=payload)

    @property
    def name(self):
        return self._name

    @property
    def action(self):
        return self._action

    @property
    def space(self):
        return self._space
```

State

The XoState class turns game information into bytes and stores it in the validator's Radix-Merkle tree, turns bytes stored in the validator's Radix-Merkle tree into game information, and does these operations with a state storage scheme that handles hash collisions.

An XO state entry consists of the UTF-8 encoding of a string with exactly four commas formatted as follows:

`<name>,<board>,<game-state>,<player-key-1>,<player-key-2>`

where

- <name> is a nonempty string not containing |,
- <board> is a string of length 9 containing only *O*, *X*, or -,
- <game-state> is one of the following: *P1-NEXT*, *P2-NEXT*, *P1-WIN*,
- *P2-WIN*, or *TIE*, and
- <player-key-1> and <player-key-2> are the (possibly empty) public keys
- associated with the game's players.

In the event of a hash collision (i.e. two or more state entries sharing the same address), the colliding state entries will stored as the UTF-8 encoding of the string `<a-entry>|<b-entry>|...`, where <a-entry>, <b-entry>,… are sorted alphabetically.

Listing 24: sawtooth_xo/processor/xo_state.py

```
XO_NAMESPACE = hashlib.sha512('xo'.encode("utf-8")).hexdigest()[0:6]


class Game(object):
    def __init__(self, name, board, state, player1, player2):
        self.name = name
        self.board = board
        self.state = state
        self.player1 = player1
        self.player2 = player2


class XoState(object):

    TIMEOUT = 3

    def __init__(self, context):
        """Constructor.
        Args:
            context (sawtooth_sdk.processor.context.Context): Access to
                validator state from within the transaction processor.
        """

        self._context = context
        self._address_cache = {}

    def delete_game(self, game_name):
        """Delete the Game named game_name from state.
        Args:
            game_name (str): The name.
        Raises:
            KeyError: The Game with game_name does not exist.
        """

        games = self._load_games(game_name=game_name)

        del games[game_name]
        if games:
            self._store_game(game_name, games=games)
```

(continues on next page)

(continued from previous page)

```
        else:
            self._delete_game(game_name)

    def set_game(self, game_name, game):
        """Store the game in the validator state.
        Args:
            game_name (str): The name.
            game (Game): The information specifying the current game.
        """

        games = self._load_games(game_name=game_name)

        games[game_name] = game

        self._store_game(game_name, games=games)

    def get_game(self, game_name):
        """Get the game associated with game_name.
        Args:
            game_name (str): The name.
        Returns:
            (Game): All the information specifying a game.
        """

        return self._load_games(game_name=game_name).get(game_name)

    def _store_game(self, game_name, games):
        address = _make_xo_address(game_name)

        state_data = self._serialize(games)

        self._address_cache[address] = state_data

        self._context.set_state(
            {address: state_data},
            timeout=self.TIMEOUT)

    def _delete_game(self, game_name):
        address = _make_xo_address(game_name)

        self._context.delete_state(
            [address],
            timeout=self.TIMEOUT)

        self._address_cache[address] = None

    def _load_games(self, game_name):
        address = _make_xo_address(game_name)

        if address in self._address_cache:
            if self._address_cache[address]:
                serialized_games = self._address_cache[address]
                games = self._deserialize(serialized_games)
            else:
                games = {}
        else:
            state_entries = self._context.get_state(
```

(continues on next page)

(continued from previous page)

```
                [address],
                timeout=self.TIMEOUT)
            if state_entries:

                self._address_cache[address] = state_entries[0].data

                games = self._deserialize(data=state_entries[0].data)

            else:
                self._address_cache[address] = None
                games = {}

        return games

    def _deserialize(self, data):
        """Take bytes stored in state and deserialize them into Python
        Game objects.
        Args:
            data (bytes): The UTF-8 encoded string stored in state.
        Returns:
            (dict): game name (str) keys, Game values.
        """

        games = {}
        try:
            for game in data.decode().split("|"):
                name, board, state, player1, player2 = game.split(",")

                games[name] = Game(name, board, state, player1, player2)
        except ValueError:
            raise InternalError("Failed to deserialize game data")

        return games

    def _serialize(self, games):
        """Takes a dict of game objects and serializes them into bytes.
        Args:
            games (dict): game name (str) keys, Game values.
        Returns:
            (bytes): The UTF-8 encoded string stored in state.
        """

        game_strs = []
        for name, g in games.items():
            game_str = ",".join(
                [name, g.board, g.state, g.player1, g.player2])
            game_strs.append(game_str)

        return "|".join(sorted(game_strs)).encode()
```

Addressing

By convention, we'll store game data at an address obtained from hashing the game name prepended with some constant.

XO data is stored in state using addresses generated from the XO family name and the name of the game being stored.

In particular, an XO address consists of the first 6 characters of the SHA-512 hash of the UTF-8 encoding of the string "xo" (which is "5b7349") plus the first 64 characters of the SHA-512 hash of the UTF-8 encoding of the game name.

For example, the XO address for a game called "my-game" could be generated as follows:

```
>>> x = hashlib.sha512('xo'.encode('utf-8')).hexdigest()[:6]
>>> x
'5b7349'
>>> y = hashlib.sha512('my-game'.encode('utf-8')).hexdigest()[:64]
>>> y
'4d4cffe9cf3fb4e41def5114a323e292af9b0e07925cca6299d671ce7fc7ec37'
>>> x+y
'5b73494d4cffe9cf3fb4e41def5114a323e292af9b0e07925cca6299d671ce7fc7ec37'
```

Addressing is implemented as follows:

```
def _make_xo_address(name):
return XO_NAMESPACE + \
    hashlib.sha512(name.encode('utf-8')).hexdigest()[:64]
```

3.7.4 Client: Building and Submitting Transactions

The process of encoding information to be submitted to a distributed ledger is generally non-trivial. A series of cryptographic safeguards are used to confirm identity and data validity. Hyperledger Sawtooth is no different, but the Python 3 SDK does provide client functionality that abstracts away most of these details, and greatly simplifies the process of making changes to the blockchain.

Creating a Private Key and Signer

In order to confirm your identity and sign the information you send to the validator, you will need a 256-bit key. Sawtooth uses the secp256k1 ECDSA standard for signing, which means that almost any set of 32 bytes is a valid key. It is fairly simple to generate a valid key using the SDK's *signing* module.

A *Signer* wraps a private key and provides some convenient methods for signing bytes and getting the private key's associated public key.

```
from sawtooth_signing import create_context
from sawtooth_signing import CryptoFactory

context = create_context('secp256k1')
private_key = context.new_random_private_key()
signer = new CryptoFactory(context).new_signer(private_key)
```

Note: This key is the **only** way to prove your identity on the blockchain. Any person possessing it will be able to sign Transactions using your identity, and there is no way to recover it if lost. It is very important that any private key is kept secret and secure.

Encoding Your Payload

Transaction payloads are composed of binary-encoded data that is opaque to the validator. The logic for encoding and decoding them rests entirely within the particular Transaction Processor itself. As a result, there are many possible formats, and you will have to look to the definition of the Transaction Processor itself for that information. As an example, the *IntegerKey* Transaction Processor uses a payload of three key/value pairs encoded as CBOR. Creating one might look like this:

```
import cbor

payload = {
    'Verb': 'set',
    'Name': 'foo',
    'Value': 42}

payload_bytes = cbor.dumps(payload)
```

Building the Transaction

Transactions are the basis for individual changes of state to the Sawtooth blockchain. They are composed of a binary payload, a binary-encoded *TransactionHeader* with some cryptographic safeguards and metadata about how it should be handled, and a signature of that header. It would be worthwhile to familiarize yourself with the information in *Transactions and Batches*, particularly the definition of TransactionHeaders.

1. Create the Transaction Header

A TransactionHeader contains information for routing a transaction to the correct transaction processor, what input and output state addresses are involved, references to prior transactions it depends on, and the public keys associated with the its signature. The header references the payload through a SHA-512 hash of the payload bytes.

```
from hashlib import sha512
from sawtooth_sdk.protobuf.transaction_pb2 import TransactionHeader

txn_header_bytes = TransactionHeader(
    family_name='intkey',
    family_version='1.0',
    inputs=['1cf1266e282c41be5e4254d8820772c5518a2c5a8c0c7f7eda19594a7eb539453e1ed7'],
    outputs=['1cf1266e282c41be5e4254d8820772c5518a2c5a8c0c7f7eda19594a7eb539453e1ed7']
    signer_public_key=signer.get_public_key().as_hex(),
    # In this example, we're signing the batch with the same private key,
    # but the batch can be signed by another party, in which case, the
    # public key will need to be associated with that key.
    batcher_public_key=signer.get_public_key().as_hex(),
    # In this example, there are no dependencies.  This list should include
    # an previous transaction header signatures that must be applied for
    # this transaction to successfully commit.
    # For example,
    # dependencies=[
↪'540a6803971d1880ec73a96cb97815a95d374cbad5d865925e5aa0432fcf1931539afe10310c122c5eaae15df61236079
↪'],
    dependencies=[],
    payload_sha512=sha512(payload_bytes).hexdigest()
).SerializeToString()
```

Note: Remember that a *batcher public_key* is the hex public key matching the private key that will later be used to sign a Transaction's Batch, and *dependencies* are the *header signatures* of Transactions that must be committed before this one (see *TransactionHeaders* in *Transactions and Batches*).

Note: The *inputs* and *outputs* are the state addresses a Transaction is allowed to read from or write to. With the Transaction above, we referenced the specific address where the value of `'foo'` is stored. Whenever possible, specific addresses should be used, as this will allow the validator to schedule transaction processing more efficiently.

Note that the methods for assigning and validating addresses are entirely up to the Transaction Processor. In the case of IntegerKey, there are specific rules to generate valid addresses, which must be followed or Transactions will be rejected. You will need to follow the addressing rules for whichever Transaction Family you are working with.

2. Create the Transaction

Once the TransactionHeader is constructed, its bytes are then used to create a signature. This header signature also acts as the ID of the transaction. The header bytes, the header signature, and the payload bytes are all used to construct the complete Transaction.

```
from sawtooth_sdk.protobuf.transaction_pb2 import Transaction

signature = signer.sign(txn_header_bytes)

txn = Transaction(
    header=txn_header_bytes,
    header_signature=signature,
    payload=payload_bytes
)
```

3. (optional) Encode the Transaction(s)

If the same machine is creating Transactions and Batches there is no need to encode the Transaction instances. However, in the use case where Transactions are being batched externally, they must be serialized before being transmitted to the batcher. The Python 3 SDK offers two options for this. One or more Transactions can be combined into a serialized *TransactionList* method, or can be serialized as a single Transaction.

```
from sawtooth_sdk.protobuf import TransactionList

txn_list_bytes = TransactionList(
    transactions=[txn1, txn2]
).SerializeToString()

txn_bytes = txn.SerializeToString()
```

Building the Batch

Once you have one or more Transaction instances ready, they must be wrapped in a *Batch*. Batches are the atomic unit of change in Sawtooth's state. When a Batch is submitted to a validator each Transaction in it will be applied (in order), or *no* Transactions will be applied. Even if your Transactions are not dependent on any others, they cannot be submitted directly to the validator. They must all be wrapped in a Batch.

1. Create the BatchHeader

Similar to the TransactionHeader, there is a *BatchHeader* for each Batch. As Batches are much simpler than Transactions, a BatchHeader needs only the public key of the signer and the list of Transaction IDs, in the same order they are listed in the Batch.

```
from sawtooth_sdk.protobuf.batch_pb2 import BatchHeader

txns = [txn]

batch_header_bytes = BatchHeader(
    signer_public_key=signer.get_public_key().as_hex(),
    transaction_ids=[txn.header_signature for txn in txns],
).SerializeToString()
```

2. Create the Batch

Using the SDK, creating a Batch is similar to creating a transaction. The header is signed, and the resulting signature acts as the Batch's ID. The Batch is then constructed out of the header bytes, the header signature, and the transactions that make up the batch.

```
from sawtooth_sdk.protobuf.batch_pb2 import Batch

signature = signer.sign(batch_header_bytes)

batch = Batch(
    header=batch_header_bytes,
    header_signature=signature,
    transactions=txns
)
```

3. Encode the Batch(es) in a BatchList

In order to submit Batches to the validator, they must be collected into a *BatchList*. Multiple batches can be submitted in one BatchList, though the Batches themselves don't necessarily need to depend on each other. Unlike Batches, a BatchList is not atomic. Batches from other clients may be interleaved with yours.

```
from sawtooth_sdk.protobuf.batch_pb2 import BatchList

batch_list_bytes = BatchList(batches=[batch]).SerializeToString()
```

Note: Note, if the transaction creator is using a different private key than the batcher, the *batcher public_key* must have been specified for every Transaction, and must have been generated from the private key being used to sign the Batch, or validation will fail.

Submitting Batches to the Validator

The prescribed way to submit Batches to the validator is via the REST API. This is an independent process that runs alongside a validator, allowing clients to communicate using HTTP/JSON standards. Simply send a *POST* request to the */batches* endpoint, with a *"Content-Type"* header of *"application/octet-stream"*, and the *body* as a serialized *BatchList*.

There are a many ways to make an HTTP request, and hopefully the submission process is fairly straightforward from here, but as an example, this is what it might look if you sent the request from the same Python 3 process that prepared the BatchList:

```
import urllib.request
from urllib.error import HTTPError

try:
    request = urllib.request.Request(
        'http://rest.api.domain/batches',
        batch_list_bytes,
        method='POST',
        headers={'Content-Type': 'application/octet-stream'})
    response = urllib.request.urlopen(request)

except HTTPError as e:
    response = e.file
```

And here is what it would look like if you saved the binary to a file, and then sent it from the command line with `curl`:

```
output = open('intkey.batches', 'wb')
output.write(batch_list_bytes)
```

```
% curl --request POST \
    --header "Content-Type: application/octet-stream" \
    --data-binary @intkey.batches \
    "http://rest.api.domain/batches"
```

3.8 Development Without an SDK

Hyperledger Sawtooth provides SDKs in a variety of languages to abstract away a great deal of complexity and simplify developing applications for the platform. The recommended way to work with Sawtooth is through one of these SDKs wherever possible. However, if there is no SDK for your language of choice, the SDK is missing some functionality, or you just want a deeper understanding of what is going on underneath the hood, this guide will cover developing for Sawtooth *without* the help of an SDK.

Note: In addition to an overview of the underlying concepts, each guide includes copious code examples in *Python 3*. This is intended simply as a concrete demonstration in a common readable language. The material covered should apply to almost any language, and actual Python development should typically happen with the *Python SDK*.

3.8.1 Building and Submitting Transactions

The process of encoding information to be submitted to a distributed ledger is generally non-trivial. A series of cryptographic safeguards are used to confirm identity and data validity, and *Hyperledger Sawtooth* is no different. SHA-512 hashes and secp256k1 signatures must be generated. Transaction and Batch Protobufs must be created and serialized. The process can be somewhat daunting, but this document will take Sawtooth client developers step by step through the process using copious Python 3 examples.

Creating Private and Public Keys

In order to sign your Transactions, you will need a 256-bit private key. Sawtooth uses the secp256k1 ECDSA standard for signing, which means that almost any set of 32 bytes is a valid key. A common way to generate one, is just to generate a random set of bytes, and then use a secp256k1 library to ensure they are valid.

For Python, the *secp256k1* module provides a *PrivateKey* handler class from which we can generate the actual bytes to use for a key.

```
import secp256k1

key_handler = secp256k1.PrivateKey()
private_key_bytes = key_handler.private_key
```

Note: This key is the **only** way to prove your identity on the blockchain. Any person possessing it will be able to sign Transactions using your identity, and there is no way to recover it if lost. It is very important that any private key is kept secret and secure.

In addition to a private key, you will need a shareable public key. It will be generated from your private key, and can be used to confirm the private key was used to sign a Transaction without exposing the private key itself. Your secp256k1 library should be able to generate a public key, and Sawtooth will expect it to be formatted as a hexadecimal string for distribution with a Transaction.

```
public_key_bytes = key_handler.public_key.serialize()

public_key_hex = public_key_bytes.hex()
```

Encoding Your Payload

Transaction payloads are composed of binary-encoded data that is opaque to the validator. The logic for encoding and decoding them rests entirely within the particular Transaction Processor itself. As a result, there are many possible formats, and you will have to look to the definition of the Transaction Processor itself for that information. As an example, the *IntegerKey* Transaction Processor uses a payload of three key/value pairs encoded as CBOR. Creating one might look like this:

```
import cbor

payload = {
    'Verb': 'set',
    'Name': 'foo',
    'Value': 42}

payload_bytes = cbor.dumps(payload)
```

Building the Transaction

Transactions are the basis for individual changes of state to the Sawtooth blockchain. They are composed of a binary payload, a binary-encoded *TransactionHeader* with some cryptographic safeguards and metadata about how it should be handled, and a signature of that header. It would be worthwhile to familiarize yourself with the information in *Transactions and Batches*, particularly the definition of TransactionHeaders.

1. Create a SHA-512 Payload Hash

However the payload was originally encoded, in order to confirm it has not been tampered with, a hash of it must be included within the Transaction's header. This hash should be created using the SHA-512 function, and then formatted as a hexadecimal string.

```
from hashlib import sha512

payload_sha512 = sha512(payload_bytes).hexdigest()
```

2. Create the TransactionHeader

Transactions and their headers are built using the Google Protocol Buffer (or Protobuf) format. This allows data to be serialized and deserialized consistently and efficiently across multiple platforms and multiple languages. The Protobuf definition files are located in the /protos directory at the root level of the sawtooth-core repo. These files must first be compiled into usable classes for your language (typically with the *protoc* command). Then, serializing a *TransactionHeader* is just a matter of plugging the right data into the right keys.

Note: Generally, to compile Python Protobufs you would follow these instructions to install Google's *Protobuf compiler* and manually compile Python classes however you like.

This example will use classes from the *Sawtooth Python SDK*, which can be compiled by running the executable script `bin/protogen`.

```
from random import randint
from sawtooth_sdk.protobuf.transaction_pb2 import TransactionHeader

txn_header = TransactionHeader(
    batcher_public_key=public_key_hex,
    # If we had any dependencies, this is what it might look like:
    # dependencies=[
↪'540a6803971d1880ec73a96cb97815a95d374cbad5d865925e5aa0432fcf1931539afe10310c122c5eaae15df61236079
↪'],
    family_name='intkey',
    family_version='1.0',
    inputs=['1cf1266e282c41be5e4254d8820772c5518a2c5a8c0c7f7eda19594a7eb539453e1ed7'],
    nonce=str(randint(0, 1000000000)),
    outputs=['1cf1266e282c41be5e4254d8820772c5518a2c5a8c0c7f7eda19594a7eb539453e1ed7
↪'],
    payload_sha512=payload_sha512,
    signer_public_key=public_key_hex)

txn_header_bytes = txn_header.SerializeToString()
```

Note: Remember that *inputs* and *outputs* are state addresses that this Transaction is allowed to read from or write to, and *dependencies* are the *header signatures* of Transactions that must be committed before this one (see TransactionHeaders in *Transactions and Batches*). The dependencies property will frequently be left empty, but generally at least one input and output must always be set, and those addresses must adhere to validation rules specific to your Transaction Family (in this case, IntegerKey).

3. Sign the Header

Once the TransactionHeader is created and serialized as a Protobuf binary, you can use your private key to create an *ECDSA signature*. In order to generate a signature the Sawtooth validator will accept, you must:

- use the *secp256k1* elliptic curve

- sign a *SHA-256* hash of the TransactionHeader binary
- use a compact 64-byte signature
- format the signature as a hexadecimal string

This is a fairly typical way to sign data, so depending on the language and library you are using, some of these steps may be handled automatically.

```
key_handler = secp256k1.PrivateKey(private_key_bytes)

# ecdsa_sign automatically generates a SHA-256 hash of the header bytes
txn_signature = key_handler.ecdsa_sign(txn_header_bytes)
txn_signature_bytes = key_handler.ecdsa_serialize_compact(txn_signature)
txn_signature_hex = txn_signature_bytes.hex()
```

4. Create the Transaction

With the other pieces in place, constructing the Transaction instance should be fairly straightforward. Create a *Transaction* class and use it to instantiate the Transaction.

```
from sawtooth_sdk.protobuf.transaction_pb2 import Transaction

txn = Transaction(
    header=txn_header_bytes,
    header_signature=txn_signature_hex,
    payload=payload_bytes)
```

5. (optional) Encode the Transaction(s)

If the same machine is creating Transactions and Batches there is no need to encode the Transaction instances. However, in the use case where Transactions are being batched externally, they must be serialized before being transmitted to the batcher. Technically any encoding scheme could be used so long as the batcher knows how to decode it, but Sawtooth does provide a *TransactionList* Protobuf for this purpose. Simply wrap a set of Transactions in the *transactions* property of a TransactionList and serialize it.

```
from sawtooth_sdk.protobuf.transaction_pb2 import TransactionList

txnList = TransactionList(transactions=[txn])
txnBytes = txnList.SerializeToString()
```

Building the Batch

Once you have one or more Transaction instances ready, they must be wrapped in a *Batch*. Batches are the atomic unit of change in Sawtooth's state. When a Batch is submitted to a validator, each Transaction in it will be applied (in order) or *no* Transactions will be applied. Even if a Transaction is not dependent on any others, it cannot be submitted directly to the validator. It must be wrapped in a Batch.

1. (optional) Decode the Transaction(s)

If the batcher is on a separate machine than the Transaction creator, any Transactions will have been encoded as a binary and transmitted. If so, they must be decoded before being wrapped in a batch. Here we assume you used a *TransactionList* to serialize the Transactions.

```
txnList = TransactionList()
txnList.ParseFromString(txnBytes)

txn = txnList.transactions[0]
```

2. Create the BatchHeader

The process for creating a *BatchHeader* is very similar to a TransactionHeader. Compile the *batch.proto* file, and then instantiate the appropriate class with the appropriate values. This time, there are just two properties: a *signer public_key*, and a set of *Transaction ids*. Just like with a TransactionHeader, the signer public_key must have been generated from the private key used to sign the Batch. The Transaction ids are a list of the *header signatures* from the Transactions to be batched. They must be in the same order as the Transactions themselves.

```
from sawtooth_sdk.protobuf.batch_pb2 import BatchHeader

batch_header = BatchHeader(
    signer_public_key=public_key_hex,
    transaction_ids=[txn.header_signature])

batch_header_bytes = batch_header.SerializeToString()
```

3. Sign the Header

The process for signing a BatchHeader is identical to signing the TransactionHeader. Create a SHA-256 hash of the header binary, use your private key to create a 64-byte secp256k1 signature, and format that signature as a hexadecimal string. As with signing a TransactionHeader, some of these steps may be handled automatically by the library you are using.

```
batch_signature = key_handler.ecdsa_sign(batch_header_bytes)

batch_signature_bytes = key_handler.ecdsa_serialize_compact(batch_signature)

batch_signature_hex = batch_signature_bytes.hex()
```

Note: The *batcher public_key* specified in every TransactionHeader must have been generated from the private key being used to sign the Batch, or validation will fail.

4. Create the Batch

Creating a *Batch* also looks a lot like creating a Transaction. Just use the compiled class to instantiate a new Batch with the proper data.

```
from sawtooth_sdk.protobuf.batch_pb2 import Batch

batch = Batch(
    header=batch_header_bytes,
    header_signature=batch_signature_hex,
    transactions=[txn])
```

5. Encode the Batch(es)

In order to submit one or more Batches to a validator, they must be serialized in a *BatchList* Protobuf. BatchLists have a single property, *batches*, which should be set to one or more Batches. Unlike Transactions, where TransactionList was a convenience, a Sawtooth validator will *only* accept Batches that have been wrapped in a BatchList.

```
from sawtooth_sdk.protobuf.batch_pb2 import BatchList

batch_list = BatchList(batches=[batch])
batch_bytes = batch_list.SerializeToString()
```

Submitting Batches to the Validator

The prescribed way to submit Batches to the validator is via the REST API. This is an independent process that runs alongside a validator, allowing clients to communicate using HTTP/JSON standards. Simply send a *POST* request to the */batches* endpoint, with a *"Content-Type"* header of *"application/octet-stream"*, and the *body* as a serialized *BatchList*.

There are a many ways to make an HTTP request, and hopefully the submission process is fairly straightforward from here, but as an example, this is what it might look if you sent the request from the same process that prepared the BatchList:

```
import urllib.request
from urllib.error import HTTPError

try:
    request = urllib.request.Request(
        'http://rest.api.domain/batches',
        batch_list_bytes,
        method='POST',
        headers={'Content-Type': 'application/octet-stream'})
    response = urllib.request.urlopen(request)

except HTTPError as e:
    response = e.file
```

And here is what it would look like if you saved the binary to a file, and then sent it from the command line with `curl`:

```
output = open('intkey.batches', 'wb')
output.write(batch_list_bytes)
```

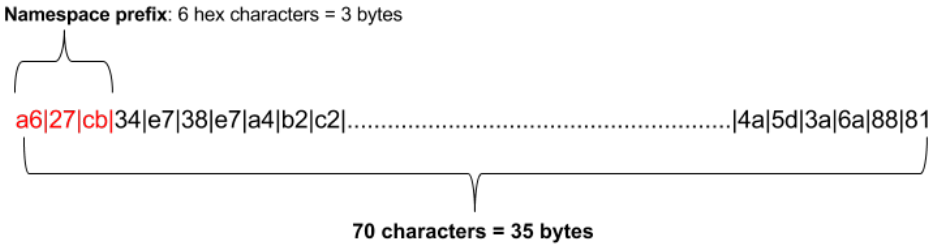
```
% curl --request POST \
    --header "Content-Type: application/octet-stream" \
    --data-binary @intkey.batches \
    "http://rest.api.domain/batches"
```

3.9 Address and Namespace Design

Hyperledger Sawtooth stores data in a Merkle-Radix tree. Data is stored in leaf nodes, and each node is accessed using an addressing scheme that is composed of 35 bytes, represented as 70 hex characters. The recommended way to construct an address is to use the hex-encoded hash values of the string or strings that make up the address elements. However, the encoding of the address is completely up to the transaction family defining the namespace, and does not need to involve hashing. Hashing is a useful way to deterministically generate likely non-colliding byte arrays of a fixed length.

3.9.1 Address Components

An address begins with a namespace prefix of six hex characters representing three bytes. The rest of the address, 32 bytes represented as 64 hex characters, can be calculated in various ways. However, certain guidelines should be followed when creating addresses, and specific requirements must be met.


Namespace prefix: 6 hex characters = 3 bytes
a6|27|cb|34|e7|38|e7|a4|b2|c2|...|4a|5d|3a|6a|88|81
70 characters = 35 bytes

The address must be deterministic: that is, any validator or client that needs to calculate the address must be able to calculate the same address, every time, when given the same inputs.

3.9.2 Namespace Prefix

All data under a namespace prefix follows a consistent address and data encoding/serialization scheme that is determined by the transaction family which defines the namespace.

The namespace prefix consists of six hex characters, or three bytes. An example namespace prefix that utilizes the string making up the transaction family namespace name to calculate the prefix is demonstrated by the following Python code:

```
prefix = hashlib.sha256("example_txn_family_namespace".encode('utf-8')).
↪hexdigest()[:6]
```

Alternatively, a namespace prefix can utilize an arbitrary scheme. The current Settings transaction family uses a prefix of ‘000000’, for example.

3.9.3 Address Construction

The rest of the address, or remaining 32 bytes (64 hex characters), must be calculated using a defined deterministic encoding format. Each address within a namespace should be unique, or the namespace consumers must be able to deal with collisions in a deterministic way.

The addressing schema can be as simple or as complex as necessary, based on the requirements of the transaction family.

Simple Example - IntegerKey

For a description of the IntegerKey Transaction family, see *IntegerKey Transaction Family*.

The transaction family prefix is:

```
hashlib.sha512('intkey'.encode('utf-8')).hexdigest()[0:6]
```

This resolves to ‘1cf126’.

To store a value in the entry *Name*, the address would be calculated like this:

```
address = "1cf126" + hashlib.sha512('name'.encode('utf-8')).hexdigest()[-64:]
```

A value could then be stored at this address, by constructing and sending a transaction to a validator, where the transaction will be processed and included in a block.

This address would also be used to retrieve the data.

3.9.4 More Complex Addressing Schemes

For a more complex example, let’s use a hypothetical transaction family which stores information on different object types for a widget. The data on each object type is keyed to a unique object identifier.

- prefix = “my-transaction-family-namespace-example”
- object-type = “widget-type”
- unique-object-identifier = ”unique-widget-identifier”

Address construction

Code Example:

```
>>> hashlib.sha256("my-transaction-family-namespace-example".encode('utf-8')).
↪hexdigest()[:6] + hashlib.sha256("widget-type".encode('utf-8')).hexdigest()[:4] +␣
↪hashlib.sha256("unique-widget-identifier".encode('utf-8')).hexdigest()[:60]
'4ae1df0ad3ac05fdc7342c50d909d2331e296badb661416896f727131207db276a908e'
```

In this case, the address is composed partly of a hexdigest made of the widget-type, and partly made up of the unique-widget-identifier. This encoding scheme choice prevents collisions between data objects that have identical identifiers, but which have different object types.

Since the addressing scheme is not mandated beyond the basic requirements, there is a lot of flexibility. The example above is just an example. Your own addressing schema should be designed with your transaction family’s requirements in mind.

Settings Transaction Family Example

See the *Settings Transaction Family* for another more complex addressing scheme.

3.10 Subscribing to Events

As blocks are committed to the blockchain, an application developer may want to receive information on events such as the creation of a new block, switching to a new fork, or application-specific events defined by a transaction family.

Client applications can subscribe to Hyperledger Sawtooth events using ZMQ and Protobuf messages. The general process is:

1. Construct a subscription that includes the event type and any filters

2. Submit the event subscription as a message to the validator
3. Wait for a response from the validator
4. Start listening for events

For information on the architecture of events in Sawtooth, see *Events and Transaction Receipts*.

3.10.1 Requirements

To subscribe to Sawtooth events, a client application can use any language that provides the following items:

- ZMQ library
- Protobuf library

In addition, the required Sawtooth protobuf messages must be compiled for the chosen language.

This section uses Python examples to show the event-subscription procedure, but the process is similar for any imperative programming language that meets these requirements.

3.10.2 Constructing a Subscription

An event subscription consists of an event type (the name of the event) and an optional list of filters for this event type.

Event Type

Each Sawtooth event has an `event_type` field that is used to determine how opaque data has been serialized and what transparent attributes to expect. An event is "in a subscription" if the event's `event_type` field matches the subscription's `event_type` field, and the event matches any filters that are in the event subscription.

The core Sawtooth events are prefixed with `sawtooth/`:

- `sawtooth/block-commit`
- `sawtooth/state-delta`

Each transaction family can define its own events. The convention is to prefix the event type with the name of the transaction family, such as `xo/create`.

Event Filters

An event subscription can include one or more event filters, which direct the validator to include events of a given type based on the event's transparent attributes. If multiple event filters are included in a subscription, only events that pass all filters will be received.

An event filter consists of a key and a match string. Filters apply only to the event attributes that have the same key as the filter. The filter's match string is compared against the attribute's value according to the type of event filter.

An event filter can either be a simple filter (an exact match to a string) or a regex filter. In addition, the filter can specify the match type of ANY (one or more) or ALL. The following filters are supported:

- `SIMPLE_ANY`
- `SIMPLE_ALL`
- `REGEX_ANY`
- `REGEX_ALL`

For example filters, see *Events*.

Event Protobuf Message

Event subscriptions are represented with the following protobuf messages, which are defined in the `events.proto` file.

```
message EventSubscription {
  string event_type = 1;
  repeated EventFilter filters = 2;
}

message EventFilter {
  string key = 1;
  string match_string = 2;

  enum FilterType {
      FILTER_TYPE_UNSET = 0;
      SIMPLE_ANY = 1;
      SIMPLE_ALL = 2;
      REGEX_ANY  = 3;
      REGEX_ALL  = 4;
    }
    FilterType filter_type = 3;
}
```

Example: Subscribing to State Deltas within a Namespace

This example constructs an event subscription for state deltas (a single change in state at a given address) within the namespace `"abcdef"`.

```
subscription = EventSubscription(
    event_type="sawtooth/state-delta",
    filters=[
        # Filter to only addresses in the "abcdef" namespace using a regex
        EventFilter(
            key="address",
            match_string="abcdef.*",
            filter_type=EventFilter.REGEX_ANY)
    ])
```

3.10.3 Submitting an Event Subscription

After constructing a subscription, send the subscription request to the validator. The following example connects to the validator's URL using ZMQ, then submits the subscription request.

```
# Setup a connection to the validator
ctx = zmq.Context()
socket = ctx.socket(zmq.DEALER)
socket.connect(url)

# Construct the request
request = ClientEventsSubscribeRequest(
```

(continues on next page)

(continued from previous page)

```
    subscriptions=[subscription]).SerializeToString()

# Construct the message wrapper
correlation_id = "123" # This must be unique for all in-flight requests
msg = Message(
    correlation_id=correlation_id,
    message_type=CLIENT_EVENTS_SUBSCRIBE_REQUEST,
    content=request)

# Send the request
socket.send_multipart([msg.SerializeToString()])
```

3.10.4 Receiving the Response

The validator will return a response indicating whether the subscription was successful. The following example receives the response and verifies the status.

```
# Receive the response
resp = socket.recv_multipart()[-1]

# Parse the message wrapper
msg = Message()
msg.ParseFromString(resp)

# Validate the response type
if msg.message_type != CLIENT_EVENTS_SUBSCRIBE_RESPONSE:
    print("Unexpected message type")

# Parse the response
response = ClientEventsSubscribeResponse()
response.ParseFromString(msg.content)

# Validate the response status
if response.status != ClientEventsSubscribeResponse.OK:
  print("Subscription failed: {}".format(response.response_message))
```

3.10.5 Listening for Events

If the event subscription was successful, events are sent to the subscriber. In order to limit network traffic, individual events are wrapped in an event list message before being sent.

The following example listens for events and prints them indefinitely.

```
while True:
  resp = socket.recv_multipart()[-1]

  # Parse the message wrapper
  msg = Message()
  msg.ParseFromString(resp)

  # Validate the response type
  if msg.message_type != CLIENT_EVENTS:
      print("Unexpected message type")
```

(continues on next page)

(continued from previous page)

```
  # Parse the response
  events = EventList()
  events.ParseFromString(msg.content)

  for event in events:
    print(event)
```

3.10.6 Correlating Events to Blocks

All events originate from some block and are only sent to the subscriber once the block is committed and state is updated. As a result, events can be treated as output from processing and committing blocks.

To match an event with the block it originated from, subscribe to the `block-commit` event. All lists of events received from the validator will contain a `block-commit` event for the block that the events came from.

Note: For forking networks, we recommend subscribing to `block-commit` events in order to watch for network forks and react appropriately. Without a subscription to `block-commit` events, there is no way to determine whether a fork has occurred.

3.10.7 Requesting Event Catch-Up

An event subscription can request "event catch-up" information on all historical events that have occurred since the creation of a specific block or blocks.

To use this feature, set the `last_known_block_ids` field in the `ClientEventsSubscribeRequest` to a list of known block ids. The validator will bring the client up to date by doing the following:

- Filter the list to include only the blocks on the current chain
- Sort the list by block number
- Send historical events from all blocks since the most recent block, one block at a time

If no blocks on the current chain are sent, the subscription will fail.

The following example submits a subscription request that includes event catch-up.

```
# Setup a connection to the validator
ctx = zmq.Context()
socket = ctx.socket(zmq.DEALER)
socket.connect(url)

# Construct the request
request = ClientEventSubscribeRequest(
    subscriptions=[subscription],
    last_known_block_ids=['000...', 'beef...'])

# Construct the message wrapper
correlation_id = "123" # This must be unique for all "in-flight requests
msg = Message(
    correlation_id=correlation_id,
    message_type=CLIENT_EVENTS_SUBSCRIBE_REQUEST,
    content=request)
```

(continues on next page)

(continued from previous page)

```
# Send the request
socket.send_multipart([msg.SerializeToString()])
```

3.10.8 Unsubscribing to Events

To unsubscribe to events, send a unsubscribe request with no arguments, then close the ZMQ socket.

This example submits an unsubscribe request.

```
# Construct the request
request = ClientEventsUnsubscribeRequest()

# Construct the message wrapper
correlation_id = "123" # This must be unique for all "in-flight requests
msg = Message(
    correlation_id=correlation_id,
    message_type=CLIENT_EVENTS_UNSUBSCRIBE_REQUEST,
    content=request)

# Send the request
socket.send_multipart([msg.SerializeToString()])
```

The following example receives the validator's response to an unsubscribe request, verifies the status, and closes the ZMQ connection.

```
# Receive the response
resp = socket.recv_multipart()[-1]

# Parse the message wrapper
msg = Message()
msg.ParseFromString(resp)

# Validate the response type
if msg.message_type != CLIENT_EVENTS_UNSUBSCRIBE_RESPONSE:
    print("Unexpected message type")

# Parse the response
response = ClientEventsUnsubscribeResponse()
response.ParseFromString(msg.content)

# Validate the response status
if response.status != ClientEventsUnsubscribeResponse.OK:
  print("Unsubscription failed: {}".format(response.response_message))

# Close the connection to the validator
socket.close()
```

CHAPTER

FOUR

TRANSACTION FAMILY SPECIFICATIONS

Sawtooth includes several transaction families as examples for developing your own transaction family.

- The *BlockInfo Transaction Family* provides a way to store information about a configurable number of historic blocks. BlockInfo is written in Python. The family name is `block_info`; the associated transaction processor is `block-info-tp`.
- The *Identity Transaction Family* is an extensible role- and policy-based system for defining permissions in a way that can be used by other Sawtooth components. Identity is written in Python. The family name is `sawtooth_identity`; the associated transaction processor is `identity-tp` (see *identity-tp*).
- The *IntegerKey Transaction Family* simply sets, increments, and decrements the value of entries stored in a state dictionary. IntegerKey is available in Go, Java, and JavaScript (Node.js). The family name is `intkey`; the associated transaction processors are `intkey-tp-go`, `intkey-tp-java`, and `intkey-tp-javascript`. The *intkey command* provides an example CLI client.
- The *Validator Registry Transaction Family* provides a way to add new validators to the network. It is used by the PoET consensus algorithm implementation to keep track of other validators. The family name is `poet_validator_registry`; the associated transaction processor is `poet-validator-registry-tp`.
- The *Settings Transaction Family* provides a methodology for storing on-chain configuration settings. Settings is written in Python. The family name is `sawtooth_settings`; the associated transaction processor is `settings-tp` (see *settings-tp*). The *sawset command* provides an example CLI client.

  Note: In a production environment, you should always run a transaction processor that supports the Settings transaction family.

- The *Smallbank Transaction Family* provides a cross-platform workload for comparing the performance of blockchain systems. Smallbank is written in Go. The family name is `smallbank`; the associated transaction processor is `smallbank-tp`.
- The *XO Transaction Family* allows two users to play a simple game of tic-tac-toe (see *Introduction to the XO Transaction Family*). XO is written in Go, JavaScript/Node.js, and Python. The family name is `xo`; the associated transaction processors are `xo-tp-go`, `xo-tp-javascript`, and `xo-tp-python`. The *xo command* provides an example CLI client.

4.1 Settings Transaction Family

4.1.1 Overview

The Settings transaction family provides a methodology for storing on-chain configuration settings.

The settings stored in state as a result of this transaction family play a critical role in the operation of a validator. For example, the consensus module uses these settings; in the case of PoET, one cross-network setting is target wait time (which must be the same across validators), and this setting is stored as sawtooth.poet.target_wait_time. Other parts of the system use these settings similarly; for example, the list of enabled transaction families is used by the transaction processing platform.

In addition, pluggable components such as transaction family implementations can use the settings during their execution.

This design supports two authorization options: a) a single authorized key which can make changes, and b) multiple authorized keys. In the case of multiple keys, a percentage of votes signed by the keys is required to make a change.

Note: While usable in a production sense, this transaction family also serves as a reference specification and implementation. The authorization scheme here provides a simple voting mechanism; another authorization scheme may be better in practice. Implementations which use a different authorization scheme can replace this implementation by adhering to the same addressing scheme and content format for settings. (For example, the location of PoET settings can not change, or the PoET consensus module will not be able to find them.)

4.1.2 State

This section describes in detail how settings are stored and addressed using the Settings transaction family.

The setting data consists of setting/value pairs. A setting is the name for the item of configuration data. The value is the data in the form of a string.

Settings

Settings are namespaced using dots:

Setting (Examples)	Value
sawtooth.poet.target_wait_time	5
sawtooth.validator.max_transactions_per_block	1000

The Settings transaction family uses the following settings for its own configuration:

Setting (Settings)	Value Description
sawtooth.settings.vote.authorized_keys	List of public keys allowed to vote
sawtooth.settings.vote.approval_threshold	Minimum number of votes required to accept or reject a proposal (default: 1)
sawtooth.settings.vote.proposals	A list of proposals to make settings changes (see note)

Note: *sawtooth.settings.vote.proposals* is a base64 encoded string of the protobuf message *SettingCandidates*. This setting cannot be modified by a proposal or a vote.

Definition of Setting Entries

The following protocol buffers definition defines setting entries:

Listing 1: File: sawtooth-core/protos/setting.proto

```
// Setting Container for the resulting state
message Setting {
    // Contains a setting entry (or entries, in the case of collisions).
    message Entry {
        string key = 1;
        string value = 2;
    }

    // List of setting entries - more than one implies a state key collision
    repeated Entry entries = 1;
}
```

sawtooth.settings.vote.proposals

The setting 'sawtooth.settings.vote.proposals' is stored as defined by the following protocol buffers definition. The value returned by this setting is a base64 encoded *SettingCandidates* message:

Listing 2: File: sawtooth-core/families/settings/protos/settings.proto

```
// Contains the vote counts for a given proposal.
message SettingCandidate {
    // An individual vote record
    message VoteRecord {
        // The public key of the voter
        string public_key = 1;

        // The voter's actual vote
        SettingVote.Vote vote = 2;

    }

    // The proposal id, a hash of the original proposal
    string proposal_id = 1;

    // The active proposal
    SettingProposal proposal = 2;

    // list of votes
    repeated VoteRecord votes = 3;
}

// Contains all the setting candidates up for vote.
message SettingCandidates {
    repeated SettingCandidate candidates = 1;
}
```

Addressing

When a setting is read or changed, it is accessed by addressing it using the following algorithm:

Setting keys are broken into four parts, based on the dots in the string. For example, the address for the key *a.b.c* is computed based on *a*, *b*, *c* and the empty string. A longer key, for example *a.b.c.d.e*, is still broken into four parts, but the remain pieces are in the last part: *a*, *b*, *c* and *d.e*.

Each of these pieces has a short hash computed (the first 16 characters of its SHA256 hash in hex) and is joined into a single address, with the settings namespace (*000000*) added at the beginning.

For example, the setting *sawtooth.settings.vote.proposals* could be set like this:

```
>>> '000000' + hashlib.sha256('sawtooth'.encode()).hexdigest()[:16] + \
    hashlib.sha256('config'.encode()).hexdigest()[:16] + \
    hashlib.sha256('vote'.encode()).hexdigest()[:16] + \
    hashlib.sha256('proposals'.encode()).hexdigest()[:16]
'000000a87cb5eafdcca6a8b79606fb3afea5bdab274474a6aa82c1c0cbf0fbcaf64c0b'
```

4.1.3 Transaction Payload

Setting transaction family payloads are defined by the following protocol buffers code:

Listing 3: File: sawtooth-core/families/settings/protos/settings.proto

```
// Setting Payload
// - Contains either a proposal or a vote.
message SettingPayload {
    // The action indicates data is contained within this payload
    enum Action {
        // A proposal action - data will be a SettingProposal
        PROPOSE = 0;

        // A vote action - data will be a SettingVote
        VOTE = 1;
    }
    // The action of this payload
    Action action = 1;

    // The content of this payload
    bytes data = 2;
}

// Setting Proposal
//
// This message proposes a change in a setting value.
message SettingProposal {
    // The setting key.  E.g. sawtooth.consensus.module
    string setting = 1;

    // The setting value. E.g. 'poet'
    string value = 2;

    // allow duplicate proposals with different hashes
    // randomly created by the client
    string nonce = 3;
}

// Setting Vote
//
// In ballot mode, a proposal must be voted on.  This message indicates an
// acceptance or rejection of a proposal, where the proposal is identified
// by its id.
message SettingVote {
    enum Vote {
```

(continues on next page)

(continued from previous page)

```
        ACCEPT = 0;
        REJECT = 1;
    }

    // The id of the proposal, as found in the
    // sawtooth.settings.vote.proposals setting field
    string proposal_id = 1;

    Vote vote = 2;
}
```

4.1.4 Transaction Header

Inputs and Outputs

The inputs for config family transactions must include:

- the address of *sawtooth.settings.vote.proposals*
- the address of *sawtooth.settings.vote.authorized_keys*
- the address of *sawtooth.settings.vote.approval_threshold*
- the address of the setting being changed

The outputs for config family transactions must include:

- the address of *sawtooth.settings.vote.proposals*
- the address of the setting being changed

Dependencies

None.

Family

- family_name: "sawtooth_settings"
- family_version: "1.0"

Encoding

The encoding field must be set to "application/protobuf".

4.1.5 Execution

Initially, the transaction processor gets the current values of *sawtooth.settings.vote.authorized_keys* from the state.

The public key of the transaction signer is checked against the values in the list of authorized keys. If it is empty, no settings can be proposed, save for the authorized keys.

A Propose action is validated. If it fails, it is considered an invalid transaction. A *proposal_id* is calculated by taking the sha256 hash of the raw *SettingProposal* bytes as they exist in the payload. Duplicate *proposal_ids* causes an invalid transaction. The proposal will be recorded in the *SettingProposals* stored in *sawtooth.settings.vote.proposals*, with one "accept" vote counted. The transaction processor outputs a *DEBUG*-level logging message similar to

```
"Adding proposal {}: {}".format(proposal_id, repr(proposal_data).
```

A Vote action is validated, checking to see if *proposal_id* exists, and the public key of the transaction has not already voted. The value of *sawtooth.settings.vote.approval_threshold* is read from the state.

- If the "accept" vote count is equal to or above the approval threshold, the proposal is applied to the state. This results in the above INFO message being logged. The proposal is deleted from the *SettingProposals* record.
- If the "reject" vote count is equal to or above the approval threshold, then it is deleted from *sawtooth.settings.vote.proposals* and an appropriate debug logging message logged.

Otherwise, the vote is recorded in the list of *sawtooth.settings.vote.proposals* by the public key and vote pair.

Validation of configuration settings is as follows:

- *sawtooth.settings.vote.approval_threshold* must be a positive integer and must be between 1 (the default) and the number of authorized keys, inclusive
- *sawtooth.settings.vote.proposals* may not be set by a proposal

4.2 Identity Transaction Family

4.2.1 Overview

On-chain permissioning is desired for transactor key and validator key permissioning. This feature requires that authorized participants for each role be stored. Currently, the only form of identification on the network is the entity's public key. Since lists of public keys are difficult to manage, an identity namespace will be used to streamline managing identities.

The identity system described here is an extensible role and policy based system for defining permissions in a way which can be utilized by other pieces of the architecture. This includes the existing permissioning components for transactor key and validator key, but in the future may also be used by transaction family implementations.

The identity namespace:

- Encompasses ways to identify participants based on public keys
- Stores a set of permit and deny rules called "policies"
- Stores the roles that those policies apply to

Policies describe a set of identities that have permission to perform some action, without specifying the logic of the action itself. Roles simply reference a policy, providing a more flexible interface. For example, a policy could be referenced by multiple roles. This way if the policy is updated, all roles referencing the policy are updated as well. An example role might be named "transactor", and references the policy that controls who is allowed to submit batches and transactions on the network. This example is further described in *Transactor Permissioning*.

Note: This transaction family will serve as a reference specification and implementation and may also be used in production. A custom implementation can be used so long as they adhere to the same addressing scheme and content format for policies and roles.

4.2.2 State

Policies

A policy will have a name and a list of entries. Each policy entry will have a type and a key list. The type will be either PERMIT_KEY or DENY_KEY and the key list will be a list of public keys.

```
// Policy will be stored in a PolicyList to account for hash collisions
message PolicyList {
  repeated Policy policies = 1;
}

message Policy {
  enum Type {
    PERMIT_KEY = 0;
    DENY_KEY = 1;
  }

  message Entry {
    // Whether this is a PERMIT_KEY or DENY_KEY entry
    Type type = 1;
    // This should be a public key or * to refer to all participants.
    string key = 2;

  }

  // name of the policy, this should be unique.
  string name = 1;

  // The entries will be processed in order from first to last.
  repeated Entry entries = 2;
}
```

Roles

A role will be made up of a role name and the name of the policy to be enforced for that role. The data will be stored in state at the address described above using the following protobuf messages:

```
// Roles will be stored in a RoleList to handle state collisions. The Roles
// will be sorted by name.
message RoleList {
  repeated Role roles = 1;
}

message Role{
  // Role name
  string name = 1;

  // Name of corresponding policy
  string policy_name = 2;
}
```

Addressing

All identity data will be stored under the special namespace of “00001d”.

For each policy, the address will be formed by concatenating the namespace, the special policy namespace of “00”, and the first 62 characters of the SHA-256 hash of the policy name:

```
>>> "00001d" + "00" + hashlib.sha256(policy_name.encode()).hexdigest()[:62]
```

Address construction for roles will follow a pattern similar to address construction in the settings namespace. Role names will be broken into four parts, where parts of the string are delimited by the “.” character. For example, the key a.b.c would be split into the parts “a”, “b”, “c”, and the empty string. If a key would have more than four parts the extra parts are left in the last part. For example, the key a.b.c.d.e would be split into “a”, “b”, “c”, and “d.e”.

A short hash is computed for each part. For the first part the first 14 characters of the SHA-256 hash are used. For the remaining parts the first 16 characters of the SHA-256 hash are used. The address is formed by concatenating the identity namespace “00001d”, the role namespace “01”, and the four short hashes.

For example, the address for the role client.query_state would be constructed as follows:

```
>>> "00001d"+ "01" + hashlib.sha256('client'.encode()).hexdigest()[:14]+ \
  hashlib.sha256('query_state'.encode()).hexdigest()[:16]+ \
  hashlib.sha256(''.encode()).hexdigest()[:16]+ \
  hashlib.sha256(''.encode()).hexdigest()[:16]
```

4.2.3 Transaction Payload

Identity transaction family payloads are defined by the following protocol buffers code:

File: sawtooth-core/families/identity/protos/identity.proto

```
message IdentityPayload {
    enum IdentityType {
      POLICY = 0;
      ROLE = 1;
    }

    // Which type of payload this is for
    IdentityType type = 1;

    // Serialize bytes of a role or a policy
    bytes data = 2;
}
```

4.2.4 Transaction Header

Inputs and Outputs

The inputs for Identity family transactions must include:

- the address of the setting *sawtooth.identity.allowed_keys*
- the address of the role or policy being changed
- if setting a role, the address of the policy to assign to the role

The outputs for Identity family transactions must include:

- the address of the role or policy being changed

Dependencies

None.

Family

- family_name: “sawtooth_identity”
- family_version: “1.0”

Encoding

The encoding field must be set to “application/protobuf”.

4.2.5 Execution

Initially, the transaction processor gets the current values of sawtooth.identity.allowed_keys from the state.

The public key of the transaction signer is checked against the values in the list of allowed keys. If it is empty, no roles or policy can be updated. If the transaction signer is not in the allowed keys the transaction is invalid.

Whether this is a role or a policy transaction is checked by looking at the `IdentityType` in the payload.

If the transaction is for setting a policy, the data in the payload will be parsed to form a `Policy` object. The `Policy` object is then checked to make sure it has a name and at least one entry. If either are missing, the transaction is considered invalid. If the policy is determined to be whole, the address for the new policy is fetched. If there is no data found at the address, a new `PolicyList` object is created, the new policy is added, and the policy list is applied to state. If there is data, it is parsed into a `PolicyList`. The new policy is added to the policy list, replacing any policy with the same name, and the policy list is applied to state.

If the transaction is for setting a role, the data in the payload will be parsed to form a `Role` object. The `Role` object is then checked to make sure it has a name and a policy_name. If either are missing, the transaction is considered invalid. The policy_name stored in the role must match a `Policy` already stored in state, if no policy is found stored at the address created by the policy_name, the transaction is invalid. If the policy exist, the address for the new role is fetched. If there is no data found at the address, a new `RoleList` object is created, the new role is added, and the policy list is applied to state. If there is data, it is parsed into a `RoleList`. The new role is added to the role list, replacing any role with the same name, and the role list is applied to state.

4.3 BlockInfo Transaction Family

4.3.1 Overview

A common feature among blockchain implementations is the ability to reference information about the blockchain while processing transactions. For example, the Ethereum Virtual Machine, which is used to process Ethereum transactions, defines a BLOCKHASH instruction which gives the processor executing the transaction access to the hashes of previous blocks.

The Blockinfo transaction family provides a way for storing information about a configurable number of historic blocks.

Note: BlockInfo transactions should only be added to a block by a BlockInfo Injector and validation rules should ensure that only one transaction of this type is included at the beginning of the block. For more information, see

Injecting Batches and On-Chain Block Validation Rules. It is important to note that the data written to state by this transaction family cannot be trusted unless both the injector and validation rules are enabled.

4.3.2 State

This section describes in detail how block information is stored and addressed using the blockinfo transaction family.

Address

The top-level namespace of this transaction family is 00b10c. This namespace is further subdivided based on the next two characters as follows:

00b10c00 Information about other namespaces; the "metadata namespace"

00b10c01 Historic block information; the "block info namespace"

Under the metadata namespace, the "zero-address" formed by concatenating the namespace and enough zeros to form a valid address will store the BlockInfoConfig.

Additional information about blocks will be stored in state under the block info namespace at an address derived from the block number. A mod is used in the calculation of the address to support chains that have a length greater than the biggest number that can be hex-encoded with 62 characters. The procedure is as follows:

1. Convert block_num to a hex string and remove the leading "0x"
2. Left pad the string with 0s until it is 62 characters long
3. Concatenate the block info namespace and the string from step 3

For example, in Python the address could be constructed with:

```
>>> '00b10c00' + hex(block_num)[2:].zfill(62))
```

BlockInfo

BlockInfoConfig will store the following information:

- The block number of the most recent block stored in state
- The block number of the oldest block stored in state
- The target number of blocks to store in state
- The network time synchronization tolerance

These values will be stored in the following protobuf message:

```
message BlockInfoConfig {
  uint64 latest_block = 1;
  uint64 oldest_block = 2;
  uint64 target_count = 3;
  uint64 sync_tolerance = 4;
}
```

Block information is stored at the address calculated above using the following protobuf message:

```
message BlockInfo {
  // Block number in the chain
  uint64 block_num = 1;

      // The header_signature of the previous block that was added to the chain.
      string previous_block_id = 2;

      // Public key for the component internal to the validator that
      // signed the BlockHeader
      string signer_public_key = 3;

      // The signature derived from signing the header
      string header_signature = 4;

      // Approximately when this block was committed, as a Unix UTC timestamp
      uint64 timestamp = 5;
}
```

4.3.3 Transaction Payload

BlockInfo transaction family payloads are defined by the following protocol buffers code:

```
message BlockInfoTxn {
  // The new block to add to state
  BlockInfo new_block = 1;

  // If this is set, the new target number of blocks to store in state
  uint64 target_count = 2;

  // If set, the new network time synchronization tolerance.
  uint64 sync_tolerance = 3;
}
```

4.3.4 Transaction Header

Inputs and Outputs

The inputs for BlockInfo family transactions must include:

- the address of the BlockInfoConfig
- the BlockInfo namespace

The outputs for BlockInfo family transactions must include:

- the address of the BlockInfoConfig
- the BlockInfo namespace

Dependencies

None.

Encoding

The encoding field must be set to 'application/protobuf'

4.3.5 Execution

Processor execution will use the following procedure:

The payload is checked to make sure it contains a valid block number, the previous block id, signer public key, and header_signature are all valid hex, and that the timestamp is greater than zero. If any of theses checks fail, the transaction is invalid.

Read the most recent block number, oldest block number, target number of blocks, and synchronization tolerance from the config zero-address.

If the config does not exist, treat this transaction as the first entry. Add the sync time and target count to the config object, along with the block_num from the payload. Add update config and new block info to state.

If the config does exist, do the following checks.

If target_count was set in the transaction, use the new value as the target number of blocks for the rest of the procedure and update the config.

If sync_tolerance was set in the transaction, use the new value as the synchronization tolerance for the rest of the procedure and update the config.

Verify the block number in the new BlockInfo message is one greater than the block number of the most recent block stored in state. If not, this transactions is invalid.

Verify the timestamp in the new BlockInfo message follows the rules below. If it does not, this transaction is invalid.

Verify the previous block id in the new BlockInfo message is equal to the block id of the most recent block stored in state. If it is not equal, the transaction is invalid.

Finally, calculate the address for the new block number. Write the new BlockInfo message to state at the address computed for that block.

If number of blocks stored in state is greater than the target number of blocks, delete the oldest BlockInfo message from state.

Write the most recent block number, oldest block number, and target number of blocks to the config zero-address.

Timestamps

Handling timestamps in a distributed network is a difficult task because peers may not have synchronized clocks. The "clock" of the network may become skewed over time, either because of peers with substantially different clocks or bad actors may have an incentive to skew the clock. If the clock of the network becomes skewed, transactions that depend on the clock may become unexpectedly invalid. If block validation depends on timestamp validation, peers may not be able to publish blocks until their clocks are adjusted to better match the network's clock.

The BlockInfo Transaction Family will use the following timestamp validation rules:

1. The timestamp in the new BlockInfo message must be greater than the timestamp in the most recent BlockInfo message in state.
2. The timestamp in the new BlockInfo message must be less than the peer's measured local time, adjusted to UTC, plus a network time synchronization tolerance that is greater than or equal to zero.

Rule 1 enforces monotonicity of timestamps. Rule 2 adds a requirement to the network that all peers be roughly synchronized. It also allows historic blocks to be validated correctly.

4.4 IntegerKey Transaction Family

4.4.1 Overview

The IntegerKey transaction family allows users to set, increment, and decrement the value of entries stored in a state dictionary.

An IntegerKey family transaction request is defined by the following values:

- A verb which describes what action the transaction takes
- A name of the entry that is to be set or changed
- A value by which the entry will be set or changed

The 'set' verb is used to create new entries. The initial value of the entry will be set to the value specified in the transaction request. The 'inc' and 'dec' verbs are used to change the value of existing entries in the state dictionary.

4.4.2 State

This section describes in detail how IntegerKey transaction information is stored and addressed.

The address values of IntegerKey transaction family entries are stored in state as a CBOR encoded dictionary, with the key being the *Name* and the value being an integer *Value*.

```
cbor.dumps({
    # The name of the entry : The current value of the entry
    'Name':'Value'
})
```

Names and *Values* must conform to the following rules:

- Valid *Names* are utf-8 encoded strings with a maximum of 20 characters
- Valid *Values* must be integers in the range of 0 through 2^{32} - 1 (32-bit unsigned int)

Addressing

IntegerKey data is stored in the state dictionary using addresses which are generated from the IntegerKey namespace prefix and the unique name of the IntegerKey entry. Addresses will adhere to the following format:

- Addresses must be a 70 character hexadecimal string
- The first 6 characters of the address are the first 6 characters of a sha512 hash of the IntegerKey namespace prefix: "intkey"
- The following 64 characters of the address are the last 64 characters of a sha512 hash of the entry *Name*

For example, an IntegerKey address could be generated as follows:

```
>>> hashlib.sha512('intkey'.encode('utf-8')).hexdigest()[0:6] + hashlib.sha512('name'.
↪encode('utf-8')).hexdigest()[-64:]
'1cf126cc488cca4cc3565a876f6040f8b73a7b92475be1d0b1bc453f6140fba7183b9a'
```

Note: Due to the possibility of hash collisions, there may be multiple *Names* in the state dictionary with the same address.

4.4.3 Transaction Payload

IntegerKey transaction request payloads are defined by the following CBOR data format:

```
cbor.dumps({
    # Describes the action the transaction takes, either 'set', 'inc', or 'dec'
    'Verb': 'verb',

    # The variable name of the entry which is to be modified
    'Name': 'name',

    # The amount to set, increment, or decrement
    'Value': 1234,
})
```

4.4.4 Transaction Header

Inputs and Outputs

The inputs for IntegerKey family transactions must include:

- Address of the *Name* being changed or added

The outputs for IntegerKey family transactions must include:

- Address of the *Name* being changed or added

Dependencies

- List of transaction *header_signatures* that are required dependencies and must be processed prior to processing this transaction

For example, the 'inc' and 'dec' transactions must list the initial 'set' transaction for the entry. If an 'inc' or 'dec' transaction is ordered before the corresponding 'set' transaction (without listing the 'set' transaction as a dependency), they will be considered invalid (because *Name* will not exist when they are processed).

Family

- family_name: "intkey"
- family_version: "1.0"

Encoding

- payload_encoding: "application/cbor"

Note: The CBOR encoding map used by IntegerKey is a definite map. For example, a transaction payload is encoded as follows:

```
>>> cbor.dumps({'Verb':'verb', 'Name':'name', 'Value':1234})
b'\xa3dVerbdverbdNamednameeValue\x19\x04\xd2'
```

CBOR Specification: RFC 7049 - Concise Binary Object Representation (CBOR)

4.4.5 Execution

The IntegerKey transaction processor receives a transaction request and a state dictionary.

If the payload of the transaction request is empty, the transaction is invalid.

The address for the transaction is generated using the algorithm stated in the Addressing section of this document. If an encoding error occurs, the transaction is invalid.

The transaction request *Verb* , *Name*, and *Value* are checked. If any of these values are empty, the transaction is invalid. *Verb* must be either 'set', 'inc', or 'dec'. *Name* must be a utf-8 encoded string with a maximum of 20 characters. *Value* must be a 32-bit unsigned integer. If any of these checks fail, the transaction is invalid.

If the *Verb* is 'set', the state dictionary is checked to determine if the *Name* associated with the transaction request already exists. If it does already exist, the transaction is invalid. Otherwise the *Name* and *Value* are stored as a new entry in the state dictionary.

If the *Verb* is 'inc', the *Name* specified by the transaction request is checked determine if the entry exists in the state dictionary. If the *Name* does not exist in the state dictionary, it is an invalid transaction. Otherwise, we attempt to increment the *Value* in the state dictionary by the *Value* specified in the transaction request. If this incrementation would result in a value outside the range of 0 through 2^{32} - 1 it is considered an invalid transaction. Otherwise, the *Value* in the state dictionary is incremented.

If the *Verb* is 'dec', the *Name* specified by the transaction request is checked determine if the entry exists in the state dictionary. If the *Name* does not exist in the state dictionary, it is an invalid transaction. Otherwise, we attempt to decrement the *Value* in the state dictionary by the *Value* specified in the transaction request. If this decrementation would result in a value outside the range of 0 through 2^{32} - 1, it is considered an invalid transaction. Otherwise, the *Value* in the state dictionary is decremented.

4.5 XO Transaction Family

4.5.1 Overview

The XO transaction family allows users to play the simple board game tic-tac-toe (also known as *Noughts and Crosses* or *X's and O's*). For more information, see *Introduction to the XO Transaction Family*.

4.5.2 State

An XO state entry consists of the UTF-8 encoding of a string with exactly four commas formatted as follows:

`<name>,<board>,<game-state>,<player-key-1>,<player-key-2>`

where

- <name> is a nonempty string not containing |,
- <board> is a string of length 9 containing only *O*, *X*, or -,
- <game-state> is one of the following: *P1-NEXT*, *P2-NEXT*, *P1-WIN*, *P2-WIN*, or *TIE*, and
- <player-key-1> and <player-key-2> are the (possibly empty) public keys associated with the game's players.

In the event of a hash collision (i.e. two or more state entries sharing the same address), the colliding state entries will stored as the UTF-8 encoding of the string `<a-entry>|<b-entry>|...`, where <a-entry>, <b-entry>,... are sorted alphabetically.

Addressing

XO data is stored in state using addresses generated from the XO family name and the name of the game being stored. In particular, an XO address consists of the first 6 characters of the SHA-512 hash of the UTF-8 encoding of the string "xo" (which is "5b7349") plus the first 64 characters of the SHA-512 hash of the UTF-8 encoding of the game name.

For example, the XO address for a game called "mygame" could be generated as follows:

```
>>> hashlib.sha512('xo'.encode('utf-8')).hexdigest()[:6] + hashlib.sha512('mygame'.
↪encode('utf-8')).hexdigest()[:64]
'5b7349700e158b598043efd6d7610345a75a00b22ac14c9278db53f586179a92b72fbd'
```

4.5.3 Transaction Payload

An XO transaction request payload consists of the UTF-8 encoding of a string with exactly two commas formatted as follows:

`<name>,<action>,<space>`

where

- <name> is a nonempty string not containing |,
- <action> is either *take* or *create*, and
- <space> is an integer strictly between 0 and 10 if <action> is *take*.

4.5.4 Transaction Header

Inputs and Outputs

The inputs and outputs for an XO transaction are just the state address generated from the transaction game name.

Dependencies

XO transactions have no explicit dependencies.

Family

- family_name: "xo"
- family_version: "1.0"

Encoding

- payload_encoding: "csv-utf8"

4.5.5 Execution

The XO transaction processor receives a transaction request and a state dictionary. If it is valid, the transaction request payload will have a game name, an action, and, if the action is *take*, a space.

1. If the action is *create*, then the transaction is invalid if the game name is already in state dictionary. Otherwise, the TP will store a new state entry with board ——— (i.e. a blank board), game state *P1-NEXT*, and empty strings for both player keys.
2. If the action is *take*, then the transaction is invalid if the game name is not in the state dictionary. Otherwise, there is a state entry under the game name with a board, game state, player-1 key, and player-2 key. The transaction is invalid if:

   - the game state is *P1-WIN*, *P2-WIN*, or *TIE*, or
   - the game state is *P1-NEXT* and the player-1 key is not null and different from the transaction signing key, or
   - the game state is *P2-NEXT* and the player-2 key is nonnull and different from the transaction signing key, or
   - the space-th character in the board-string is not -.

   Otherwise, the transaction processor will update the state entry according to the following rules:

   (a) Player keys - If the player-1 key is null (the empty string), it will be updated to key with which the transaction was signed. If the player-1 key is nonnull and the player-2 key is null, the player-2 will be updated to the signing key. Otherwise, the player keys will not be changed.

   (b) Board - If the game state is *P1-NEXT*, the updated board will be the same as the initial board except with the space-th character replaced by the character *X*, and the same for the state *P2-NEXT* and the character *O*.

   (c) Game state - Call the first three characters of the board string its first row, the next three characters its second row, and the last three characters its third row.If any of the rows consist of the same character, say that character has a *win* on the board. If all the rows have the same first character or the same second character or the same third character (the board's columns), that character has a win on the board. If the first character of the first row and the second character of the second row and the third character of the third row are the same, or if the third character of the first row and the second character of the second row and the first character of the third row are the same, that character has a win on the board.

      - If *X* has a win on the board and *O* doesn't, the updated state will be *P1-WINS*.
      - If *O* has a win on the board and *X* doesn't, the updated state will be *P2-WINS*.
      - Otherwise, if the updated board doesn't contain - (if the board has no empty spaces), the updated state will be *TIE*.
      - Otherwise, the updated state will be *P2-NEXT* if the initial state is *P1-NEXT* and *P1-NEXT* and if the initial state is *P2-NEXT*.

4.6 Validator Registry Transaction Family

4.6.1 Overview

The validator registry transaction family provides a way to add new validators to the network.

The validator's information is used by poet_consensus to verify that when a validator tries to claim a block it is following the block claiming policies of PoET consensus. For example, the Z policy will refuse blocks from a validator if it has already claimed a disproportionate amount of blocks compared to the other validators on the network. The C policy says that a validator may not claim blocks, after being added to the registry or updating their information, until a certain number of blocks are claimed by other participants. The number of blocks claimed since signup can be found by checking the current block number against the block number that is stored in the validator information when the validator registry information has either been added or updated. And finally the K policy requires new signup information to be sent to the validator registry after it has claimed a maximum number of block. If the validator is not found in the validator registry the block will be rejected since there is no way to validate.

Currently the number of blocks claimed by a specific validator is stored within the poet_consensus, not within the validator registry.

4.6.2 State

This section describes in detail how validator information, including identification and signup data, is stored and addressed using the validator registry transaction family.

The following protocol buffers definition defines the validator info:

```
message ValidatorInfo {
  // The name of the endpoint. It should be a human readable name.
  string name = 1;

  // The validator's public key (currently using signer_public_key as this is
  // stored in the transaction header)
  string id = 2;

  // This is the protocol buffer described below.
  SignUpInfo signup_info = 3;

  // The header signature for the ValidatorRegistryPayload transaction. This
  // will be used to look up the block the ValidatorInfo was committed on.
  string transaction_id = 4;
}
```

The above ValidatorInfo includes the following protocol buffer:

```
message SignUpInfo {
  // Encoded public key corresponding to private key used by PoET to sign
  // wait certificates.
  string poet_public_key = 1;

  // Information that can be used internally to verify the validity of
  // the signup information stored as an opaque buffer.
  string proof_data = 2;

  // A string corresponding to the anti-Sybil ID for the enclave that
  // generated the signup information.
  string anti_sybil_id = 3;

  // The nonce associated with the signup info.  Note that this must match
  // the nonce provided when the signup info was created.
  string nonce = 4;
}
```

Currently the poet_enclave needs a way to check how many validators are currently registered within the validator registry. It also needs a way to find the validator_id associated with an anti-sybil_id. A ValidatorMap, where the anti_sybil_id is the key and the validator_id is the value solves both of these problems.

```
message ValidatorMap {
    // Contains a validator entry where the key is an  anti_sybil_id,
    // and the value is a validator_id
    message Entry {
```

(continues on next page)

(continued from previous page)

```
        string key = 1;
        string value = 2;
    }

    // List of validator entries
    repeated Entry entries = 1;
}
```

This can be used to check the number of validators registered. This information is used to decide what number of blocks a validator has to wait for before it can start claiming blocks after it adds new signup information to the validator registry. This check is necessary because if the number of blocks that must be waited on is greater than the number of validators minus one, it is possible for the network to get into a state where nobody can publish blocks because the validators are all waiting for more blocks to be committed or their signup information to be added to a block.

Validator registry transaction would not be able to be done at the same time as any other transaction as an update to the ValidatorMap is necessary. However all other transaction that need to access the state set by the validator registry, can be done in parallel since it will only be a read and the statics for each validator is stored in the poet_enclave. If this was changed so that the stats were stored in the validator registry this would require a write to state every time a block is published and would reduce the ability for parallelism.

Addressing

When a validator's signup info is registered or updated it should be accessed using the following algorithm:

Addresses for the validator registry transaction family are set by adding sha256 hash of the validator's id to the validator registry namespace. The namespace for the validator registry will be the first 6 characters of the sha256 hash of the string "validator_registry", which is "6a4372" For example, the validator signup info of a validator with the id "validator_id" could be set like this:

```
>>>"6a4372" + hashlib.sha256('validator_id'.encode("utf-8")).hexdigest()
'6a43722aee5b550a3cbd1595f4de10049ee805bc035b5e232dfacfc31cc6275170b30d'
```

The map of the current registered validator_id should be stored at the following address:

```
>>>"6a4372" + hashlib.sha256('validator_map'.encode()).hexdigest()
'6a437247a1c12c0fb03aa6e242e6ce988d1cdc7fcc8c2a62ab3ab1202325d7d677e84c'
```

4.6.3 Transaction Payload

Validator registry transaction family payloads are defined by the following protocol buffers code:

```
message ValidatorRegistryPayload {
  // The action that the transaction processor will take. Currently this
  // is only "register", but could include other actions in the futures
  // such as "revoke"
  string verb = 1;

  // The name of the endpoint. It should be a human readable name.
  string name = 2;

  // Validator's public key (currently using signer_public_key as this is
  // stored in the transaction header)
```

(continues on next page)

(continued from previous page)

```
  string id = 3;

  // This is the protocol buffer described above.
  SignUpInfo signup_info = 4;

}
```

4.6.4 Transaction Header

Inputs and Outputs

The inputs for validator registry family transactions must include:

- the address of *validator_id*
- the address of *validator_map*
- the address of *sawtooth.poet.report_public_key_pem*
- the address of *sawtooth.poet.valid_enclave_measurement*
- the address of *sawtooth.poet.valid_enclave_basenames*

The outputs for validator registry family transactions must include:

- the address of *validator_id*
- the address of *validator_map*

Dependencies

None

Family

- family_name: “sawtooth_validator_registry”
- family_version: “1.0”

Encoding

The encoding field must be set to “application/protobuf”.

4.6.5 Execution

Untrusted Python code that is a part of the transaction processor will verify the attestation verification report for the signup information. It is important to note that the IAS report public key will need to be on the blockchain, and that it will need to be set during configuration. This will allow both the simulator logic and real SGX logic to be the same.

If the validator_name does not match syntactic requirements, the transaction is invalid. The current requirement is that the validator_name is 64 characters or less.

If the validator_id does not match the transaction signer, the transaction is invalid. The validator_id should be the same as the signer_public_key stored in the transaction header.

The signup info needs to be verified next. The signup info, public key hash, and the most recent wait_certificate_id are passed to verify the signup data.

If any of the signup info checks fail verification, the validator_registry transaction is rejected and a invalid transaction response is returned.

If the transaction is deemed to be valid, the validator_id is used to find the address where the validator_info should be stored. Store the serialized ValidatorInfo protocol buffer in state at the address as mentioned above. If this validator is new (not updating its SignUpInfo), the validator's id needs to be added to the validator_map.

4.7 Smallbank Transaction Family

4.7.1 Overview

The Smallbank transaction family is based on the H-Store Smallbank benchmark originally published as part of the H-Store database benchmarking project.

> http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

This transaction family is intended for use in benchmarking and performance testing activities in the hopes of providing a somewhat cross-platform workload for comparing the performance of blockchain systems.

The Smallbank Transaction Family consists of a single Account datatype, one transaction which creates accounts and five transactions which modify account values based on a set of rules. The H-Store benchmark included a sixth transaction for reading balances which does not make sense in the context of blockchain transactions.

Smallbank Transaction payloads consist of a serialized protobuf wrapper containing the type and data payload of the sub transaction types.

4.7.2 State

This section describes in detail how Smallbank transaction information is stored and addressed.

Smallbank Account values are stored in state as serialized protobufs containing generated customer IDs, customer names, and savings and checking balances. The 'primary key' of the data is the unique customer_id.

```
message Account {
    // Customer ID
    uint32 customer_id = 1;

    // Customer Name
    string customer_name = 2;

    // Savings Balance (in cents to avoid float)
    uint32 savings_balance = 3;

    // Checking Balance (in cents to avoid float)
    uint32 checking_balance = 4;
}
```

Addressing

Smallbank Account data is stored in state using addresses which are generated from the Smallbank namespace prefix and the unique customer_id of the Account entry. Addresses will adhere to the following format:

- Addresses must be a 70 character hexadecimal string
- The first 6 characters of the address are the first 6 characters of a sha512 hash of the Smallbank namespace prefix: "smallbank"
- The following 64 characters of the address are the last 64 characters of a sha512 hash of the value of customer_id

For example, a Smallbank address could be generated as follows:

```
>>> customer_id = 42
>>> hashlib.sha512('smallbank'.encode('utf-8')).hexdigest()[0:6] + hashlib.
↪sha512(str(customer_id).encode('utf-8')).hexdigest()[-64:]
'3325143ff98ae73225156b2c6c9ceddbfc16f5453e8fa49fc10e5d96a3885546a46ef4'
```

4.7.3 Transaction Payload

Smallbank transaction request payloads are defined by the following protobuf structure:

```
message SmallbankTransactionPayload {
    enum Type {
        CREATE_ACCOUNT = 1;
        DEPOSIT_CHECKING = 2;
        WRITE_CHECK = 3;
        TRANSACT_SAVINGS = 4;
        SEND_PAYMENT = 5;
        AMALGAMATE = 6;
    }
    PayloadType payload_type = 1;
    CreateAccountTransactionData create_account = 2;
    DepositCheckingTransactionData deposit_checking = 3;
    WriteCheckTranasctionData write_check = 4;
    TransactSavingsTransactionData transact_savings = 5;
    SendPaymentTransactionData send_payment = 6;
    AmalgamateTransactionData amalgamate = 7;
}
```

Based on the selected type, the data field will contain the appropriate transaction data (these messages would be defined within SmallbankTransactionPayload):

```
message CreateAccountTransactionData {
    // The CreateAccountTransaction creates an account

    // Customer ID
    uint32 customer_id = 1;

    // Customer Name
    string customer_name = 2;

    // Initial Savings Balance (in cents to avoid float)
    uint32 initial_savings_balance = 3;

    // Initial Checking Balance (in cents to avoid float)
    uint32 initial_checking_balance = 4;
}

message DepositCheckingTransactionData {
    // The DepositCheckingTransction adds an amount to the customer's
```

(continues on next page)

(continued from previous page)

```
    // checking account.

    // Customer ID
    uint32 customer_id = 1;

    // Amount
    uint32 amount = 2;
}

message WriteCheckTransactionData {
    // The WriteCheckTransaction removes an amount from the customer's
    // checking account.

    // Customer ID
    uint32 customer_id = 1;

    // Amount
    uint32 amount = 2;
}

message TransactSavingsTransactionData {
    // The TransactSavingsTransaction adds an amount to the customer's
    // savings account. Amount may be a negative number.

    // Customer ID
    uint32 customer_id = 1;

    // Amount
    int32 amount = 2;
}

message SendPaymentTransactionData {
    // The SendPaymentTransaction transfers an amount from one customer's
    // checking account to another customer's checking account.

    // Source Customer ID
    uint32 source_customer_id = 1;

    // Destination Customer ID
    uint32 dest_customer_id = 2;

    // Amount
    uint32 amount = 3;
}

message AmalgamateTransactionData {
    // The AmalgamateTransaction transfers the entire contents of one
    // customer's savings account into another customer's checking
    // account.

    // Source Customer ID
    uint32 source_customer_id = 1;

    // Destination Customer ID
    uint32 dest_customer_id = 2;
}
```

4.7.4 Transaction Header

Inputs and Outputs

The inputs for Smallbank family transactions must include:

- Address of the customer_id being accessed for CreateAccount, DepositChecking, WriteCheck, and TransactSavings transactions, and both the source and destination customer_ids being accessed for SendPayment and Amalgamate transactions.

The outputs for Smallbank family transactions must include:

- Address of the customer_id being modified for CreateAccount, DepositChecking, WriteCheck, and TransactSavings transactions, and both the source and destination customer_ids being modified for SendPayment and Amalgamate transactions.

Dependencies

- List of transaction *header_signatures* that are required dependencies and must be processed prior to processing this transaction

Note: While any CreateAccount transaction signatures should probably be listed in any other modification transactions that reference those accounts, it may be sufficient to submit a set of CreateAccount transactions, ensure they are committed to the chain and then proceed without explicit dependencies.

Family

- family_name: "smallbank"
- family_version: "1.0"

Encoding

- payload_encoding: "application/protobuf"

4.7.5 Execution

A CreateAccount transaction is only valid if:

- customer_id is specified
- there is not already an existing account at the address associated with that customer_id
- customer_name is specified (not an empty string)
- initial_savings_balance is specified
- initial_checking_balance is specified

The result of a successful CreateAccount transaction is that the new Account object is set in state.

A DepositChecking transaction is only valid if:

- customer_id is specified
- there is an account at the address associated with that customer_id

- amount is specified
- amount + Account.checking_balance doesn't result in an overflow of uint32

The result of a successful DepositChecking transaction is that the specified Account.checking_balance = Account.checking_balance + amount.

A WriteCheck transaction is only valid if:

- customer_id is specified
- there is an account at the address associated with that customer_id
- amount is specified
- Account.checking_balance - amount doesn't result in a value < 0

The result of a successful WriteCheck transaction is that the specified Account's checking balance is decremented by amount.

A TransactSavings transaction is only valid if:

- customer_id is specified
- there is an account at the address associated with that customer_id
- amount is specified
- Account.savings_balance + amount doesn't result in a value < 0 or a value which overflows uint32

The result of a successful TransactSavings transaction is that the specified Account's savings balance is modified by the addition of amount (which may be negative).

A SendPayment transaction is only valid if:

- both source_customer_id and dest_customer_id are specified
- there is an account at both locations
- source.checking_balance - amount doesn't result in a value < 0
- dest.checking_balance + amount doesn't result in a value which overflows uint32

The result of a successful SendPayment transaction is that the source's checking balance is decremented by amount and the destination's checking balance is incremented by amount.

An Amalgamate transaction is only valid if:

- both source_customer_id and dest_customer_id are specified
- there is an account at both locations
- source.savings_balance is > 0
- dest.checking_balance + source.savings_balance does not overflow uint32

The result of a successful Amalgamate transaction is that the destination's checking balance is incremented by the value of the source's savings balance and that the source's savings balance is set to zero.

CHAPTER

FIVE

SYSTEM ADMINISTRATOR'S GUIDE

5.1 Installing Hyperledger Sawtooth

The easiest way to install Hyperledger Sawtooth is with apt-get.

Note: These instructions have been tested on Ubuntu 16.04 only.

Stable Builds:

To add the stable repository, run these commands in a terminal window on your host system:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys␣
↪8AA7AF1F1091A5FD
$ sudo add-apt-repository 'deb http://repo.sawtooth.me/ubuntu/1.0/stable xenial␣
↪universe'
```

Nightly Builds:

If you'd like the latest version of Sawtooth, we also provide a repository of nightly builds. These builds may incorporate undocumented features and should be used for testing purposes only. To use the nightly repository, run the following commands in a terminal window on your host system:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys␣
↪44FC67F19B2466EA
$ sudo apt-add-repository "deb http://repo.sawtooth.me/ubuntu/nightly xenial universe"
```

Update your package lists and install Sawtooth:

```
$ sudo apt-get update
$ sudo apt-get install -y sawtooth
```

5.2 Running Sawtooth as a Service

When installing Sawtooth using apt-get, *systemd* units are added for the following components. These can then be started, stopped, and restarted using the *systemctl* command:

- validator
- transaction processors
- rest_api

To learn more about *systemd* and the *systemctl* command, check out this guide:

5.2.1 Viewing Console Output

To view the console output that you would see if you ran the components manually, run the following command:

```
$ sudo journalctl -f \
    -u sawtooth-validator \
    -u sawtooth-settings-tp \
    -u sawtooth-poet-validator-registry-tp \
    -u sawtooth-rest-api
```

Validator Start-up Process

5.2.2 Create Genesis Block

The first validator created in a new network must load a genesis block on creation to enable other validators to join the network. Prior to starting the first validator, run the following commands to generate a genesis block that the first validator can load:

```
$ sawtooth keygen --key-dir ~/sawtooth
$ sawset genesis --key ~/sawtooth.priv
$ sawadm genesis config-genesis.batch
```

5.2.3 Running Sawtooth

Note: Before starting the `validator` component you may need to generate the validator keypairs using the following command:

```
$ sudo sawadm keygen
```

To start a component using *systemd*, run the following command where *COMPONENT* is one of:

- validator
- rest-api
- intkey-tp-python
- settings-tp
- xo-tp-python

```
$ sudo systemctl start sawtooth-COMPONENT
```

To see the status of a component run:

```
$ sudo systemctl status sawtooth-COMPONENT
```

Likewise, to stop a component run:

```
$ sudo systemctl stop sawtooth-COMPONENT
```

5.3 Configuring Sawtooth

Each Sawtooth component, such as the validator or the REST API, has an optional configuration file that controls the component's behavior. You can also specify configuration options on the command line when starting the component. For more information, see *CLI Command Reference*.

When a Sawtooth component starts, it looks for a TOML configuration file in the config directory (`config_dir`). By default, configuration files are stored in `/etc/sawtooth`; see *Path Configuration File* for more information on the config directory location.

In addition, the Sawtooth log output can be configured with a log config file in TOML or YAML format. By default, Sawtooth stores error and debug log messages for each component in the log directory. For more information, see *Log Configuration*.

The following sections describe each component's configuration file.

5.3.1 Validator Configuration File

The validator configuration file specifies network information that allows the validator to advertise itself properly and search for peers. This file also contains settings for optional authorization roles and transactor permissions.

If the config directory contains a file named `validator.toml`, the configuration settings are applied when the validator starts. (By default, the config directory is `/etc/sawtooth/`; see *Path Configuration File* for more information.) Specifying an option on the command line overrides the setting in the configuration file.

An example configuration file is in `/sawtooth-core/packaging/validator.toml.example`. To create a validator configuration file, copy the example file to the config directory and name it `validator.toml`. Then edit the file to change the example configuration options as necessary for your system.

Note: See *Using Sawtooth with the SGX Implementation of PoET* for an example of changing the settings in `validator.toml` when configuring Sawtooth with the SGX implementation of PoET.

The `validator.toml` configuration file has the following options:

- `bind` = ["`endpoint`", "`endpoint`"]

  Sets the network and component endpoints. Default network bind interface: `tcp://127.0.0.1:8800`. Default component bind interface: `tcp://127.0.0.1:4004`.

  Each string has the format `{option}:{endpoint}`, where `{option}` is either `network` or `component`. For example:

```
bind = [
  "network:tcp://127.0.0.1:8800",
  "component:tcp://127.0.0.1:4004"
]
```

- `peering = "{static,dynamic}"`

  Specifies the type of peering approach the validator should take: static or dynamic. Default: `static`.

  Static peering attempts to peer only with the candidates provided with the peers option. For example:

```
peering = "static"
```

  Dynamic peering first processes any static peers, starts topology buildouts, then uses the URLs specified by the seeds option for the initial connection to the validator network.

```
peering = "dynamic"
```

- `endpoint = "URL"`

  Sets the advertised network endpoint URL. Default: tcp://127.0.0.1:8800. Replace the external interface and port values with either the publicly addressable IP address and port or with the NAT values for your validator. For example:

```
endpoint = "tcp://127.0.0.1:8800"
```

- `seeds` = [`URI`]

  (Dynamic peering only.) Specifies the URI or URIs for the initial connection to the validator network. Specify multiple URIs in a comma-separated list; each URI must be enclosed in double quotes. Default: none.

  Note that this option is not needed in static peering mode.

  Replace the seed address and port values with either the publicly addressable IP address and port or with the NAT values for the other nodes in your network. For example:

```
seeds = ["tcp://127.0.0.1:8801"]
```

- `peers` = ["*URL*"]

  Specifies a static list of peers to attempt to connect to. Default: none.

```
peers = ["tcp://127.0.0.1:8801"]
```

- `scheduler` = '*type*'

  Determines the type of scheduler to use: serial or parallel. Default: `serial`. For example:

```
scheduler = 'serial'
```

- `network_public_key` and `network_private_key`

  Specifies the curve ZMQ key pair used to create a secured network based on side-band sharing of a single network key pair to all participating nodes. Default: none.

  Enclose the key in single quotes; for example:

```
network_public_key = 'wFMwoOt>yFqI/ek.G[tfMMILHWw#vXB[Sv}>l>i)'
network_private_key = 'r&oJ5aQDj4+V]p2:Lz70Eu0x#m%IwzBdP(}&hWM*'
```

Important: If these options are not set or the configuration file does not exist, the network will default to being insecure.

- `opentsdb_url` = "*value*"

  Sets the host and port for Open TSDB database (used for metrics). Default: none.

- `opentsdb_db` = "*name*"

  Sets the name of the Open TSDB database. Default: none.

- `opentsdb_username` = *username*

  Sets the username for the Open TSDB database. Default: none.

- `opentsdb_password` = *password*

  Sets the password for the Open TSDB database. Default: none.

- `network = "{trust,challenge}"`

  Specifies the type of authorization that must be performed for the different type of authorization roles on the network: trust or challenge. Default: trust.

  This option must be in the `[roles]` section of the file. For example:

  ```
  [roles]
  network = "trust"
  ```

  For more information, see *Authorization Types*.

- "*role*" = "*policy*"

  Sets the off-chain transactor permissions for the role or roles that specify which transactors are allowed to sign batches on the system. Multiple roles can be defined, using one "*role*" = "*policy*" entry per line. Default: none.

  The role names specified in this config file must match the roles stored in state for transactor permissioning. For example:

  - `transactor`
  - `transactor.transaction_signer`
  - `transactor.transaction_signer.{tp_name}`
  - `transactor.batch_signer`

  For *policy*, specify a policy file in `policy_dir` (by default, `/etc/sawtooth/`). Each policy file contains permit and deny rules for the transactors; see *Off-Chain Transactor Permissioning*.

  Because transactor roles and policy files can have a period in the name, use double-quotes so that TOML can process these settings. For example:

  ```
  [permissions]
  "transactor" = "policy.example"
  "transactor.transaction_signer" = "policy.example"
  ```

  Note: The `default` role cannot be set in the configuration file. Use the `sawtooth identity` command to change this on-chain-only setting.

  See *Configuring Permissions* for more information on roles and permissions.

- `minimum_peer_connectivity` = *min*

  The minimum number of peers required before stopping peer search. Default: 3 For example:

  ```
  minimum_peer_connectivity = 3
  ```

- `maximum_peer_connectivity` = *max*

  The maximum number of peers that will be accepted. Default: 10. For example:

  ```
  maximum_peer_connectivity = 10
  ```

5.3.2 REST API Configuration File

The REST API configuration file specifies network connection settings and an optional timeout value.

If the config directory contains a file named `rest_api.toml`, the configuration settings are applied when the REST API starts. (By default, the config directory is `/etc/sawtooth/`; see *Path Configuration File* for more information.) Specifying a command-line option will override the setting in the configuration file.

An example configuration file is in `/sawtooth-core/rest_api/packaging/rest_api.toml.example`. To create a REST API configuration file, copy the example file to the config directory and name it `rest_api.toml`. Then edit the file to change the example configuration options as necessary for your system.

The `rest_api.toml` configuration file has the following options:

- `bind` = ["*HOST:PORT*"]

  Sets the port and host for the REST API to run on. Default: `127.0.0.1:8008`. For example:

  ```
  bind = ["127.0.0.1:8008"]
  ```

- `connect` = "*URL*"

  Identifies the URL of a running validator. Default: `tcp://localhost:4004`. For example:

  ```
  connect = "tcp://localhost:4004"
  ```

- `timeout` = *value*

  Specifies the time, in seconds, to wait for a validator response. Default: 300. For example:

  ```
  timeout = 900
  ```

- `opentsdb_url` = "*value*"

  Sets the host and port for Open TSDB database (used for metrics).

- `opentsdb_db` = "*name*"

  Sets the name of the Open TSDB database. Default: none.

- `opentsdb_username` = *username*

  Sets the username for the Open TSDB database. Default: none.

- `opentsdb_password` = *password*

  Sets the password for the Open TSDB database. Default: none.

5.3.3 Sawtooth CLI Configuration File

The Sawtooth CLI configuration file specifies arguments to be used by the `sawtooth` command and its subcommands. For example, you can use this file to set the URL of the REST API once, rather than entering the `--url` option for each subcommand.

If the config directory contains a file named `cli.toml`, the configuration settings are applied when the `sawtooth` command is run. (By default, the config directory is /etc/sawtooth/; see *Path Configuration File* for more information.) Specifying command-line options will override the settings in the configuration file.

An example configuration file is in `/sawtooth-core/cli/cli.toml.example`. To create a CLI configuration file, copy the example file to the config directory and name it `cli.toml`.

The example file shows the format of the `url` option. To use it, uncomment the line and replace the default value with the actual URL for the REST API.

```
# The REST API URL
#   url = "http://localhost:8008"
```

5.3.4 PoET SGX Enclave Configuration File

This configuration file specifies configuration settings for a PoET SGX enclave.

If the config directory contains a file named `poet_enclave_sgx.toml`, the configuration settings are applied when the component starts. (By default, the config directory is `/etc/sawtooth/`; see *Path Configuration File* for more information.) Specifying a command-line option will override the setting in the configuration file.

An example configuration file is in `/sawtooth-core/consensus/poet/sgx/packaging/poet_enclave_sgx.toml.example`. To create a PoET SGX enclave configuration file, copy the example file to the config directory and name it `poet_enclave_sgx.toml`. Then edit the file to change the example configuration options as necessary for your system.

Note: See *Using Sawtooth with the SGX Implementation of PoET* for an example of changing settings in `poet_enclave_sgx.toml` when configuring Sawtooth with the SGX implementation of PoET.

The `poet_enclave_sgx.toml` configuration file has the following options:

- `spid` = '*string*'

  Specifies the Service Provider ID (SPID), which is linked to the key pair used to authenticate with the attestation service. Default: none. The SPID value is a 32-digit hex string tied to the enclave implementation; for example:

  ```
  spid = 'DEADBEEF00000000DEADBEEF00000000'
  ```

- `ias_url` = '*URL*'

  Specifies the URL of the Intel Attestation Service (IAS) server. Default: none. Note that the URL shown in `poet_enclave_sgx.toml.example` is an example server for debug enclaves only:

  ```
  ias_url = 'https://test-as.sgx.trustedservices.intel.com:443'
  ```

- `spid_cert_file` = '*/full/path/to/certificate.pem*'

  Identifies the PEM-encoded certificate file that was submitted to Intel in order to obtain a SPID. Default: none. Specify the full path to the certificate file. This pem file can be created from `cert.crt` and `cert.key` files with this command:

  ```
  $ cat cert.crt cert.key > cert.pem
  ```

  Or from `cert.pfx` file with following command:

  ```
  $ openssl pkcs12 -in cert.pfx -out cert.pem -nodes
  ```

5.3.5 Path Configuration File

Important: Changing the path settings in this file is usually unnecessary. For non-standard directory paths, use the `SAWTOOTH_HOME` environment variable instead of this configuration file.

The path.toml configuration changes the Sawtooth data_dir directory. This file should be used only when installing on an operating system distribution where the default paths are not appropriate. For example, some Unix-based operating systems do not use /var/lib, so it would be appropriate to use this file to set data_dir to the natural operating system default path for application data.

This file configures the following settings:

- key_dir = *path*

  Directory path to use when loading key files

- data_dir = *path*

  Directory path for storing data files such as the block store

- log_dir = *path*

  Directory path to use to write log files (by default, an error log and a debug log; see *Log Configuration*).

- policy_dir = *path*

  Directory path for storing policies

The default directory paths depend on whether the SAWTOOTH_HOME environment variable is set. When SAWTOOTH_HOME is set, the default paths are:

- key_dir = SAWTOOTH_HOME/keys/
- data_dir = SAWTOOTH_HOME/data/
- log_dir = SAWTOOTH_HOME/logs/
- policy_dir = SAWTOOTH_HOME/policy/

For example, if SAWTOOTH_HOME is set to /tmp/testing, the default path for data_dir is /tmp/testing/data/.

When SAWTOOTH_HOME is not set, the operating system defaults are used. On Linux, the default path settings are:

- key_dir = /etc/sawtooth/keys
- data_dir = /var/lib/sawtooth
- log_dir = /var/log/sawtooth
- policy_dir = /etc/sawtooth/policy

Sawtooth also uses config_dir to determine the directory path containing the configuration files. Note that this directory is fixed; it cannot be changed in the path.toml configuration file.

- If SAWTOOTH_HOME is set, conf_dir = SAWTOOTH_HOME/etc/
- If SAWTOOTH_HOME is not set, conf_dir = /etc/sawtooth

5.3.6 Identity Transaction Processor Configuration File

The Identity transaction processor configuration file specifies the validator endpoint connection to use.

If the config directory contains a file named identity.toml, the configuration settings are applied when the transaction processor starts. Specifying a command-line option will override the setting in the configuration file.

Note: By default, the config directory is /etc/sawtooth/. See *Path Configuration File* for more information.

An example configuration file is in /sawtooth-core/families/identity/packaging/identity.toml.example. To create a Identity transaction processor configuration file, copy the example file to the config directory and name it `identity.toml`. Then edit the file to change the example configuration options as necessary for your system.

The `identity.toml` configuration file has the following option:

- `connect` = "*URL*"

  Identifies the URL of a running validator. Default: `tcp://localhost:4004`. For example:

  ```
  connect = "tcp://localhost:4004"
  ```

5.3.7 Settings Transaction Processor Configuration File

The Settings transaction processor configuration file specifies the validator endpoint connection to use.

If the config directory contains a file named `settings.toml`, the configuration settings are applied when the transaction processor starts. Specifying a command-line option will override the setting in the configuration file.

Note: By default, the config directory is /etc/sawtooth/. See *Path Configuration File* for more information.

An example configuration file is in `/sawtooth-core/families/settings/packaging/settings.toml.example`. To create a Settings transaction processor configuration file, copy the example file to the config directory and name it `settings.toml`. Then edit the file to change the example configuration options as necessary for your system.

The `settings.toml` configuration file has the following option:

- `connect` = "*URL*"

  Identifies the URL of a running validator. Default: `tcp://localhost:4004`. For example:

  ```
  connect = "tcp://localhost:4004"
  ```

5.3.8 XO Transaction Processor Configuration File

The XO transaction processor configuration file specifies the validator endpoint connection to use.

If the config directory contains a file named `xo.toml`, the configuration settings are applied when the transaction processor starts. Specifying a command-line option will override the setting in the configuration file.

Note: By default, the config directory is /etc/sawtooth/. See *Path Configuration File* for more information.

An example configuration file is in `/sawtooth-core/families/xo/packaging/xo.toml.example`. To create a XO transaction processor configuration file, copy the example file to the config directory and name it `xo.toml`. Then edit the file to change the example configuration options as necessary for your system.

The `xo.toml` configuration file has the following option:

- `connect` = "*URL*"

  Identifies the URL of a running validator. Default: `tcp://localhost:4004`. For example:

  ```
  connect = "tcp://localhost:4004"
  ```

5.3.9 Log Configuration

Overview

The validator and the Python SDK make it easy to customize the logging output. This is done by creating a log config file in TOML or YAML format and passing it to the built-in Python logging module.

Note: Use YAML to configure a remote syslog service. Due to a limitation in the TOML spec, you cannot configure a remote syslog service using TOML.

Log Files

If there is no log configuration file provided, the default is to create an error log and a debug log. Theses files will be stored in the log directory (`log_dir`) in a location determined by the `SAWTOOTH_HOME` environment variable. For more information, see *Path Configuration File*.

The names of the validator log files are:

- `validator-debug.log`
- `validator-error.log`

For Python transaction processors, the author determines the name of the log file. It is highly encouraged that the file names are unique for each running processor to avoid naming conflicts. The example transaction processors provided with the SDK uses the following naming convention:

- `{TPname}-{zmqID}-debug.log`
- `{TPname}-{zmqID}-error.log`

Examples:

- `intkey-18670799cbbe4367-debug.log`
- `intkey-18670799cbbe4367-error.log`

Log Configuration

To change the default logging behavior of a Sawtooth component, such as the validator, put a log configuration file in the config directory (see *Path Configuration File*).

The validator log config file should be named `log_config.toml`.

Each transaction processor can define its own config file. The name of this file is determined by the author. The transaction processors included in the Python SDK use the following naming convention:

- `{TransactionFamilyName}_log_config.toml`

For example, the IntegerKey (`intkey`) log configuration file is `intkey_log_config.toml`.

Examples

Configure a Specific Logger

If the default logs give too much information, you can configure a specific logger that will only report on the area of the code you are interested in.

This example `log_config.toml` file creates a handler that only writes interconnect logs to the directory and file specified.

```
version = 1
disable_existing_loggers = false

[formatters.simple]
format = "[%(asctime)s.%(msecs)03d [%(threadName)s] %(module)s %(levelname)s]
 ↪%(message)s"
datefmt = "%H:%M:%S"

[handlers.interconnect]
level = "DEBUG"
formatter = "simple"
class = "logging.FileHandler"
filename = "path/filename.log"

[loggers."sawtooth_validator.networking.interconnect"]
level = "DEBUG"
propagate = true
handlers = [ "interconnect"]
```

The formatter and log level can also be specified to provide the exact information you want in your logs.

Rotating File Handler

Below is an example of how to setup rotating logs. This is useful when the logs may grow very large, such as with a long-running network. For example:

```
 [formatters.simple]
 format = "[%(asctime)s.%(msecs)03d [%(threadName)s] %(module)s %(levelname)s]
↪%(message)s"
 datefmt = "%H:%M:%S"

 [handlers.interconnect]
 level = "DEBUG"
 formatter = "simple"
 class = "logging.handlers.RotatingFileHandler"
 filename = "example-interconnect.log"
 maxBytes = 50000000
 backupCount=20

[loggers."sawtooth_validator.networking.interconnect"]
 level = "DEBUG"
 propagate = true
 handlers = [ "interconnect"]
```

If one file exceeds the `maxBytes` set in the config file, that file will be renamed to `filename.log.1` and a new `filename.log` will be written to. This process continues for the number of files plus one set in the `backupCount`. After that point, the file that is being written to is rotated. The current file being written to is always `filename.log`.

For more Python configuration options, see the Python documentation at https://docs.python.org/3/library/logging.config.html.

5.4 Using a Proxy Server to Authorize the REST API

As a lightweight shim on top of internal communications, requests sent to the *Hyperledger Sawtooth* REST API are simply passed on to the validator, without any sort of authorization. While this is in keeping with the public nature of blockchains, that behavior may not be desirable in every use case. Rather than internally implementing one authorization scheme or another, the REST API is designed to work well behind a proxy server, allowing any available authorization scheme to be implemented externally.

5.4.1 Forwarding URL Info with Headers

For the most part putting the REST API behind a proxy server should just work. The majority of what it does will work equally well whether or not it is communicating directly with a client. The notable exceptions being the *"link"* parameter sent back in the response envelope, and the *"previous"* and *"next"* links that are sent back with a paging response. These URLs can be a convenience for clients, but not when crucial URL information is destroyed by proxying. In that case, a link that should look like this:

```
{
  "link": "https://hyperledger.org/sawtooth/blocks?head=..."
}
```

Might instead look like this:

```
{
  "link": "http://localhost:8008/blocks?head=..."
}
```

The solution to this problem is sending the destroyed information using HTTP request headers. The Sawtooth REST API will properly recognize and parse two sorts of headers.

"X-Forwarded" Headers

Although they aren't part of any standard, the various *X-Forwarded* headers are a very common way of communicating useful information about a proxy. There are three of these headers that the REST API may look for when building links.

header	description	example
X-Forwarded-Host	The domain name of the proxy server.	*hyperledger.org*
X-Forwarded-Proto	The protocol/scheme used to make request.	*https*
X-Forwarded-Path	An uncommon header implemented specially by the REST API to handle extra path information. Only necessary if the proxy endpoints do not map directly to the REST API endpoints (i.e. *"hyperledger.org/sawtooth/blocks"* -> *"localhost:8008/blocks"*).	*/sawtooth*

"Forwarded" Header

Although less common, the same information can be sent using a single *Forwarded* header, standardized by RFC7239. The Forwarded header contains semi-colon-separated key value pairs. It might for example look like this:

```
Forwarded: for=196.168.1.1; host=proxy1.com, host=proxy2.com; proto="https"
```

There are three keys in particular the REST API will look for when building response links:

key	description	example
host	The domain name of the proxy server.	*host=hyperledger.org*
proto	The protocol/scheme used to make request.	*proto=https*
path	An non-standard key header used to handle extra path information. Only necessary if the proxy endpoints do not map directly to the REST API endpoints (i.e. *"hyperledger.org/sawtooth/blocks"* -> *"localhost:8008/blocks"*).	*path="/sawtooth"*

Note: Any key in a *Forwarded* header can be set multiple times, each instance comma-separated, allowing for a chain of proxy information to be traced. However, the REST API will always reference the leftmost of any particular key. It is only interested in producing an accurate link for the original client.

5.4.2 Apache Proxy Setup Guide

For further clarification, this section walks through the setup of a simple Apache 2 proxy server secured with Basic Auth and https, pointed at an instance of the *Sawtooth* REST API.

Install Apache

We'll begin by installing Apache and its components. These commands may require `sudo`.

```
$ apt-get update
$ apt-get install -y apache2
$ a2enmod ssl
$ a2enmod headers
$ a2enmod proxy_http
```

Set Up Passwords and Certificates

First we'll create a password file for the user *"sawtooth"*, with the password *"sawtooth"*. You can generate other .htpasswd files as well, just make sure to authorize those users in the config file below.

```
$ echo "sawtooth:\$apr1\$cyAIkitu\$Cv6M2hHJlNgnVvKbUdlFr." >/tmp/.password
```

Then we'll use `openssl` to build a self-signed SSL certificate. This certificate will not be good enough for most HTTP clients, but is suitable for testing purposes.

```
$ openssl req -x509 -nodes -days 7300 -newkey rsa:2048 \
    -subj /C=US/ST=MN/L=Mpls/O=Sawtooth/CN=sawtooth \
    -keyout /tmp/.ssl.key \
    -out /tmp/.ssl.crt
```

Configure Proxy

Now we'll set up the proxy by editing an Apache config files. This may require `sudo`.

```
$ vi /etc/apache2/sites-enabled/000-default.conf
```

Edit the file to look like this:

```
<VirtualHost *:443>
    ServerName sawtooth
    ServerAdmin sawtooth@sawtooth
    DocumentRoot /var/www/html

    SSLEngine on
    SSLCertificateFile /tmp/.ssl.crt
    SSLCertificateKeyFile /tmp/.ssl.key
    RequestHeader set X-Forwarded-Proto "https"

    <Location />
        Options Indexes FollowSymLinks
        AllowOverride None
        AuthType Basic
        AuthName "Enter password"
        AuthUserFile "/tmp/.password"
        Require user sawtooth
        Require all denied
    </Location>
</VirtualHost>

ProxyPass /sawtooth http://localhost:8008
ProxyPassReverse /sawtooth http://localhost:8008
RequestHeader set X-Forwarded-Path "/sawtooth"
```

Note: Apache will automatically set the *X-Forwarded-Host* header.

Start Apache, a Validator, and the REST API

Start or restart Apache as appropriate. This may require `sudo`.

```
$ apachectl start
```

```
$ apachectl restart
```

Start a validator, and the REST API.

```
$ sawadm keygen
$ sawadm genesis
$ sawtooth-validator -v --endpoint localhost:8800
$ sawtooth-rest-api -v
```

Send Test Requests

Finally, let's use `curl` to make some requests and make sure everything worked. We'll start by querying the REST API directly:

```
$ curl http://localhost:8008/blocks
```

The response link should look like this:

```
{
  "link": "http://localhost:8008/blocks?head=..."
}
```

You should also be able to get back a `401` by querying the proxy without authorization:

```
$ curl https://localhost/sawtooth/blocks --insecure
```

Note: The `--insecure` flag just forces curl to complete the request even though there isn't an official SSL Certificate. It does *not* bypass Basic Auth.

And finally, if we send a properly authorized request:

```
$ curl https://localhost/sawtooth/blocks --insecure -u sawtooth:sawtooth
```

We should get back a response that looks very similar to querying the REST API directly, but with a new *link* that reflects the URL we sent the request to:

```
{
  "link": "https://localhost/sawtooth/blocks?head=..."
}
```

5.5 Configuring Permissions

5.5.1 Transactor Permissioning

Overview

A running protected network needs a mechanism for limiting which transactors are allowed to submit batches and transactions to the validators. There are two different methods for defining the transactors a validator will accept.

The first method is configuring a validator to only accept batches and transactions from predefined transactors that are loaded from a local validator config file. Once the validator is configured the list of allowed transactors is immutable while the validator is running. This set of permissions are only enforced when receiving a batch from a connected client, but not when receiving a batch from a peer on the network.

The second method uses the identity namespace which will allow for network-wide updates and agreement on allowed transactors. The allowed transactors are checked when a batch is received from a client, received from a peer, and when a block is validated.

When using both on-chain and off-chain configuration, the validator only accepts transactions and batches from a client if both configurations agree it should be accepted. If a batch is received from a peer, only on-chain configuration is checked.

It is possible that in the time that the batch has been in the pending queue, the transactor permissions have been updated to no longer allow a batch signer or a transaction signer that is being included in a block. For this reason, the allowed transactor roles will also need to be checked on block validation.

Off-Chain Transactor Permissioning

Validators can be locally configured by creating a validator.toml file and placing it in the config directory. Adding configuration for transactor permissioning to the configuration shall be done in the following format:

validator.toml:

```
[permissions]

ROLE = POLICY_NAME
```

Where ROLE is one of the roles defined below for transactor permissioning and equals the filename for a policy. Multiple roles may be defined in the same format within the config file.

The policies are stored in the policy_dir defined by the path config. Each policy will be made up of permit and deny rules, similar to policies defined in the Identity Namespace. Each line will contain either a “PERMIT_KEY” or “DENY_KEY” and should be followed with either a public key for an allowed transactor or an * to allow all possible transactors. The rules will be evaluated in order.

Policy file:

```
PERMIT_KEY <key>
DENY_KEY <key>
```

On-Chain Transactor Permissioning

The Identity Namespace stores roles as key-value pairs, where the key is a role name and the value is a policy. All roles that limit who is allowed to sign transactions and batches should start with transactor as a prefix.

```
transactor.SUB_ROLE = POLICY_NAME
```

SUB_ROLEs are more specific signing roles, for example who is allowed to sign transactions. Each role equals a policy name that corresponds to a policy stored in state. The policy contains a list of public keys that correspond to the identity signing key of the transactors.

To configure on-chain roles, the signer of identity transactions needs to have their public key set in the Setting “sawtooth.identity.allowed_keys”. Only those transactors whose public keys are in that setting are allowed to update roles and policies.

```
$ sawset proposal create sawtooth.identity.allowed_
↪keys=02b2be336a6ada8f96881cd55fd848c10386d99d0a05e1778d2fc1c60c2783c2f4
```

Once your signer key is stored in the setting, the `identity-tp` command can be used to set and update roles and policies. Make sure that the Identity transaction processor and the REST API are running.

```
usage: sawtooth identity policy create [-h] [-k KEY] [-o OUTPUT | --url URL]
                                       [--wait WAIT]
                                       name rule [rule ...]

Creates a policy that can be set to a role or changes a policy without
resetting the role.

positional arguments:
  name                  name of the new policy
  rule                  rule with the format "PERMIT_KEY <key>" or "DENY_KEY
                        <key> (multiple "rule" arguments can be specified)
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     specify the signing key for the resulting batches
  -o OUTPUT, --output OUTPUT
                        specify the output filename for the resulting batches
  --url URL             identify the URL of a validator's REST API
  --wait WAIT           set time, in seconds, to wait for the policy to commit
                        when submitting to the REST API.
```

```
usage: sawtooth identity role create [-h] [-k KEY] [--wait WAIT]
                                     [-o OUTPUT | --url URL]
                                     name policy

Creates a new role that can be used to enforce permissions.

positional arguments:
  name                  name of the role
  policy                identify policy that role will be restricted to

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     specify the signing key for the resulting batches
  --wait WAIT           set time, in seconds, to wait for a role to commit
                        when submitting to the REST API.
  -o OUTPUT, --output OUTPUT
                        specify the output filename for the resulting batches
  --url URL             the URL of a validator's REST API
```

For example, running the following will create a policy that permits all and is named policy_1:

```
$ sawtooth identity policy create policy_1 "PERMIT_KEY *"
```

To see the policy in state, run the following command:

```
$ sawtooth identity policy list
policy_1:
  Entries:
    PERMIT_KEY *
```

Finally, run the following command to set the role for transactor to permit all:

```
$ sawtooth identity role create transactor policy_1
```

To see the role in state, run the following command:

```
$ sawtooth identity role list
transactor: policy_1
```

Transactor Roles

The following are the identity roles that are used to control which transactors are allowed to sign transactions and batches on the system.

default: When evaluating role permissions, if the role has not been set, the default policy will be used. The policy can be changed to meet the network's requirements after initial start-up by submitting a new policy with the name default. If the default policy has not been explicitly set, the default is "PERMIT_KEY *".

transactor: The top level role for controlling who can sign transactions and batches on the system will be transactor. This role shall be used when the allowed transactors for transactions and batches are the same. Any transactor whose public key is in the policy will be allowed to sign transactions and batches, unless a more specific sub-role disallows their public key.

transactor.transaction_signer: If a transaction is received that is signed by a transactor who is not permitted by the policy, the batch containing the transaction will be dropped.

transactor.transaction_signer.{tp_name}: If a transaction is received for a specific transaction family that is signed by a transactor who is not permitted by the policy, the batch containing the transaction will be dropped.

transactor.batch_signer: If a batch is received that is signed by a transactor who is not permitted by the policy, that batch will be dropped.

5.5.2 Validator Key Permissioning

Overview

One of the permissioning requirements is that the validator network be able to limit the nodes that are able to connect to it. The permissioning rules determine the roles a connection is able to play on the network. The roles control the types of messages that can be sent and received over a given connection. The entities acting in the different roles will be referred to as requesters below.

Validators are able to determine whether messages delivered to them should be handled or dropped based on a set of role and identities stored within the Identity namespace. Each requester will be identified by the public key derived from their identity signing key. Permission verifiers examine incoming messages against the policy and the current configuration and either permit, drop, or respond with an error. In certain cases, the connection will be forcibly closed – for example: if a node is not allowed to connect to the validator network.

This on-chain approach allows the whole network to change its policies at the same time while the network is live, instead of relying on a startup configuration.

Configuring Authorization

The Identity namespace stores roles as key-value pairs, where the key is a role name and the value is a policy. Validator network permissioning roles use the following pattern:

```
network[.SUB_ROLE] = POLICY_NAME
```

where network is the name of the role to be used for validator network permissioning. POLICY_NAME refers to the name of a policy that is set in the Identity namespace. The policy defines the public keys that are allowed to participate in that role. The policy is made up of PERMIT_KEY and DENY_KEY rules and is evaluated in order. If the public key is denied, the connection will be rejected. For more information, please look at the *Identity Transaction Family*.

Like above, run the following command to set the role for network to permit all:

```
$ sawtooth identity role create network policy_1
```

To see the role in state, run the following command:

```
$ sawtooth identity role list
network: policy_1
```

Network Roles

The following is the suggested high-level role for on-chain validator network permissioning.

network If a validator receives a peer request from a node whose public key is not permitted by the policy, the message will be dropped, an AuthorizationViolation will be returned, and the connection will be closed.

This role is checked by the permission verifier when the following messages are received:

- GossipMessage
- GetPeersRequest
- PeerRegisterRequest
- PeerUnregisterRequest
- GossipBlockRequest
- GossipBlockResponse
- GossipBatchByBatchIdRequest
- GossipBatchByTransactionIdRequest
- GossipBatchResponse
- GossipPeersRequest
- GossipPeersResponse

network.consensus If a validator receives a GossipMessage that contains a new block published by a node whose public key is not permitted by the policy, the message will be dropped, an AuthorizationViolation will be returned, and the connection will be closed.

In the future, other sub-roles can be added to further restrict access to specific functions of the role. For example network.gossip could control who is allowed to send gossip messages.

5.6 Using Sawtooth with the SGX Implementation of PoET

Note: These instructions have been tested on Ubuntu 16.04 only.

5.6.1 Prerequisites

BIOS Update

Important: You may need to update your BIOS with a security fix before running Hyperledger Sawtooth with PoET. Affected versions and instructions for updating can be found on Intel's website. If you're running an affected version, you must update the BIOS to ensure that these installation instructions work correctly.

You can verify the BIOS version after the machine has booted by running:

```
$ sudo lshw| grep -A5 *-firmware
     *-firmware
          description: BIOS
          vendor: Intel Corp.
          physical id: 0
          version: BNKBL357.86A.0050.2017.0816.2002
          date: 08/16/2017
```

5.6.2 Install SGX/PSW

Install the prerequisites for SGX/PSW:

```
$ sudo apt-get update &&
  sudo apt-get install -q -y \
      alien \
      autoconf \
      automake \
      build-essential \
      cmake \
      libcurl4-openssl-dev \
      libprotobuf-dev \
      libssl-dev \
      libtool \
      libxml2-dev \
      ocaml \
      pkg-config \
      protobuf-compiler \
      python \
      unzip \
      uuid-dev \
      wget
```

Download and install the SGX driver:

```
$ mkdir ~/sgx && cd ~/sgx
$ wget https://download.01.org/intel-sgx/linux-2.0/sgx_linux_x64_driver_eb61a95.bin
$ chmod +x sgx_linux_x64_driver_eb61a95.bin
$ sudo ./sgx_linux_x64_driver_eb61a95.bin
```

Download and install the Intel Capability Licensing Client. This is presently available only as an .rpm, so you must convert it to a .deb package with alien:

```
$ wget http://registrationcenter-download.intel.com/akdlm/irc_nas/11414/iclsClient-1.
↪45.449.12-1.x86_64.rpm
$ sudo alien --scripts iclsClient-1.45.449.12-1.x86_64.rpm
$ sudo dpkg -i iclsclient_1.45.449.12-2_amd64.deb
```

Download and install the Dynamic Application Loader Host Interface (JHI):

```
$ wget https://github.com/01org/dynamic-application-loader-host-interface/archive/
↪master.zip -O jhi-master.zip
$ unzip jhi-master.zip && cd dynamic-application-loader-host-interface-master
$ cmake .
$ make
$ sudo make install
$ sudo systemctl enable jhi
```

Download and install the Intel SGX Platform Software (PSW):

```
$ cd ~/sgx
$ wget https://download.01.org/intel-sgx/linux-2.0/sgx_linux_ubuntu16.04.1_x64_psw_2.
↪0.100.40950.bin
$ chmod +x sgx_linux_ubuntu16.04.1_x64_psw_2.0.100.40950.bin
$ sudo ./sgx_linux_ubuntu16.04.1_x64_psw_2.0.100.40950.bin
```

Check to make sure the kernel module is loaded:

```
$ lsmod | grep sgx
isgx                   36864  2
```

If the output does not show the isgx module, make sure that SGX is set to “Enabled” in the BIOS.

If you’re still having trouble, the SGX software may need to be reinstalled:

```
$ sudo /opt/intel/sgxpsw/uninstall.sh
$ cd ~/sgx
$ sudo ./sgx_linux_x64_driver_eb61a95.bin
$ sudo ./sgx_linux_ubuntu16.04.1_x64_psw_2.0.100.40950.bin
```

After ensuring that the SGX kernel module is loaded, go to the next section to install and configure Sawtooth.

5.6.3 Configuring Sawtooth to Use SGX

Install Sawtooth

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys␣
↪8AA7AF1F1091A5FD
$ sudo add-apt-repository 'deb http://repo.sawtooth.me/ubuntu/1.0/stable xenial␣
↪universe'
$ sudo apt-get update
$ sudo apt-get install -y -q \
  sawtooth \
  python3-sawtooth-poet-sgx
```

Certificate File

The configuration process requires an SGX certificate file in PEM format (.pem), which you will need before continuing.

Instructions for creating your own service provider certificate can be found here.

After your certificate is created you’ll need to register it with the attestation service. Click here for the registration form.

Configure the Validator to Use SGX PoET

After installing Sawtooth, add config settings so PoET will work properly.

Create the file /etc/sawtooth/poet_enclave_sgx.toml with your favorite editor (such as vi):

```
$ sudo vi /etc/sawtooth/poet_enclave_sgx.toml
```

Add the following lines, replacing [example] with the spid value provided by Intel:

```
# Service Provider ID. It is linked to the key pair used to authenticate with
# the attestation service.

spid = '[example]'

# ias_url is the URL of the Intel Attestation Service (IAS) server.

ias_url = 'https://test-as.sgx.trustedservices.intel.com:443'

# spid_cert_file is the full path to the PEM-encoded certificate file that was
# submitted to Intel in order to obtain a SPID

spid_cert_file = '/etc/sawtooth/sgx-certificate.pem'
```

Next, install the .pem certificate file that you downloaded earlier. Replace [example] in the path below with the path to the certificate file on your local system:

```
$ sudo install -o root -g sawtooth -m 640 \
/[example]/sgx-certificate.pem /etc/sawtooth/sgx-certificate.pem
```

Create validator keys:

```
$ sudo sawadm keygen
```

Note: If you're configuring multiple validators, the steps below are required for the first validator only. For additional validators, you can skip the rest of this procedure. Continue with *Change the Validator Config File*.

Become the sawtooth user and change to `/tmp`. In the following commands, the prompt `[sawtooth@system]` shows the commands that must be executed as the sawtooth user.

```
$ sudo -u sawtooth -s
[sawtooth@system]$ cd /tmp
```

Create a genesis batch:

```
[sawtooth@system]$ sawset genesis --key /etc/sawtooth/keys/validator.priv -o config-
↪genesis.batch
```

Create and submit a proposal:

```
[sawtooth@system]$ sawset proposal create -k /etc/sawtooth/keys/validator.priv \
sawtooth.consensus.algorithm=poet \
sawtooth.poet.report_public_key_pem="$(cat /etc/sawtooth/ias_rk_pub.pem)" \
sawtooth.poet.valid_enclave_measurements=$(poet enclave --enclave-module sgx␣
↪measurement) \
sawtooth.poet.valid_enclave_basenames=$(poet enclave --enclave-module sgx basename) \
sawtooth.poet.enclave_module_name=sawtooth_poet_sgx.poet_enclave_sgx.poet_enclave \
-o config.batch
```

When the `sawset proposal` command runs, you should see several lines of output showing that the SGX enclave has been initialized:

```
[12:03:58 WARNING poet_enclave] SGX PoET enclave initialized.
[12:03:59 WARNING poet_enclave] SGX PoET enclave initialized.
```

Note: There's quite a bit going on in the previous `sawset proposal` command, so let's take a closer look at what it accomplishes:

`sawtooth.consensus.algorithm=poet` Changes the consensus algorithm to PoET.

`sawtooth.poet.report_public_key_pem="$(cat /etc/sawtooth/ias_rk_pub.pem)"` Adds the public key that the validator registry transaction processor uses to verify attestation reports.

`sawtooth.poet.valid_enclave_measurements=$(poet enclave --enclave-module sgx measurement)` Adds the enclave measurement for your enclave to the blockchain for the validator registry transaction processor to use to check signup information.

`sawtooth.poet.valid_enclave_basenames=$(poet enclave --enclave-module sgx basename)` Adds the enclave basename for your enclave to the blockchain for the validator registry transaction processor to use to check signup information.

`sawtooth.poet.enclave_module_name` Specifies the name of the Python module that implements the PoET enclave. In this case, `sawtooth_poet_sgx.poet_enclave_sgx.poet_enclave` is the SGX version of the enclave; it includes the Python code as well as the Python extension.

Create a poet-genesis batch:

```
[sawtooth@system]$ poet registration create -k /etc/sawtooth/keys/validator.priv \
  --enclave-module sgx -o poet_genesis.batch
Writing key state for PoET public key: 0387a451...9932a998
Generating poet_genesis.batch
```

Create a genesis block:

```
[sawtooth@system]$ sawadm genesis config-genesis.batch config.batch poet_genesis.batch
```

You'll see some output indicating success:

```
Processing config-genesis.batch...
Processing config.batch...
Processing poet_genesis.batch...
Generating /var/lib/sawtooth/genesis.batch
```

Genesis configuration is complete! Log out of the sawtooth account:

```
[sawtooth@system]$ exit
$
```

Change the Validator Config File

You must specify some networking information so that the validator advertises itself properly and knows where to search for peers. Create the file `/etc/sawtooth/validator.toml`:

```
$ sudo vi /etc/sawtooth/validator.toml
```

Add the following content to the file:

```
#
# Hyperledger Sawtooth -- Validator Configuration
#
```

(continues on next page)

(continued from previous page)

```
# This file should exist in the defined config directory and allows
# validators to be configured without the need for command line options.

# The following is a possible example.

# Bind is used to set the network and component endpoints. It should be a list
# of strings in the format "option:endpoint", where the options are currently
# network and component.
bind = [
  "network:tcp://eno1:8800",
  "component:tcp://127.0.0.1:4004"
]

# The type of peering approach the validator should take. Choices are 'static'
# which only attempts to peer with candidates provided with the peers option,
# and 'dynamic' which will do topology buildouts. If 'dynamic' is provided,
# any static peers will be processed first, prior to the topology buildout
# starting.
peering = "dynamic"

# Advertised network endpoint URL.
endpoint = "tcp://[external interface]:[port]"

# Uri(s) to connect to in order to initially connect to the validator network,
# in the format tcp://hostname:port. This is not needed in static peering mode
# and defaults to None.
seeds = ["tcp://[seed address 1]:[port]",
         "tcp://[seed address 2]:[port]"]

# A list of peers to attempt to connect to in the format tcp://hostname:port.
# It defaults to None.
peers = []

# The type of scheduler to use. The choices are 'serial' or 'parallel'.
scheduler = 'serial'

# A Curve ZMQ key pair are used to create a secured network based on side-band
# sharing of a single network key pair to all participating nodes.
# Note if the config file does not exist or these are not set, the network
# will default to being insecure.
#network_public_key = ''
#network_private_key = ''
```

Next, locate the `endpoint` section in this file. Replace the external interface and port values with either the publicly addressable IP address and port or the NAT values for your validator.

```
endpoint = "tcp://[external interface]:[port]"
```

Find the `seeds` section in the config file. Replace the seed address and port values with either the publicly addressable IP address and port or the NAT values for the other nodes in your network.

```
seeds = ["tcp://[seed address 1]:[port]",
         "tcp://[seed address 2]:[port]"]
```

If necessary, change the network bind interface in the `bind` section.

```
bind = [
  "network:tcp://eno1:8800",
  "component:tcp://127.0.0.1:4004"
]
```

The default network bind interface is "eno1". If this device doesn't exist on your machine, change the `network` definition to specify the correct bind interface.

Important: If the bind interface doesn't exist, you may see a ZMQ error in the sawtooth-validator systemd logs when attempting to start the validator, as in this example:

```
Jun 02 14:50:37 ubuntu validator[15461]:   File "/usr/lib/python3.5/threading.py",␣
↪line 862, in run
...
Jun 02 14:50:37 ubuntu validator[15461]:   File "zmq/backend/cython/socket.pyx", line␣
↪487, in zmq.backend.cython.socket.Socket.bind (zmq/backend/cython/socket.c:5156)
Jun 02 14:50:37 ubuntu validator[15461]:   File "zmq/backend/cython/checkrc.pxd",␣
↪line 25, in zmq.backend.cython.checkrc._check_rc (zmq/backend/cython/socket.c:7535)
Jun 02 14:50:37 ubuntu validator[15461]: zmq.error.ZMQError: No such device
Jun 02 14:50:37 ubuntu systemd[1]: sawtooth-validator.service: Main process exited,␣
↪code=exited, status=1/FAILURE
Jun 02 14:50:37 ubuntu systemd[1]: sawtooth-validator.service: Unit entered failed␣
↪state.
Jun 02 14:50:37 ubuntu systemd[1]: sawtooth-validator.service: Failed with result
↪'exit-code'.
```

Restrict permssions on `validator.toml` to protect the network private key.

```
$ sudo chown root:sawtooth /etc/sawtooth/validator.toml
$ sudo chown 640 /etc/sawtooth/validator.toml
```

Start the Sawtooth Services

Use these commands to start the Sawtooth services:

```
$ sudo systemctl start sawtooth-rest-api.service
$ sudo systemctl start sawtooth-poet-validator-registry-tp.service
$ sudo systemctl start sawtooth-validator.service
$ sudo systemctl start sawtooth-settings-tp.service
$ sudo systemctl start sawtooth-intkey-tp-python.service
```

You can follow the logs by running:

```
$ sudo journalctl -f \
-u sawtooth-validator \
-u sawtooth-tp_settings \
-u sawtooth-poet-validator-registry-tp \
-u sawtooth-rest-api \
-u sawtooth-intkey-tp-python
```

Additional logging output can be found in `/var/log/sawtooth/`.

To verify that the services are running:

```
$ sudo systemctl status sawtooth-rest-api.service
$ sudo systemctl status sawtooth-poet-validator-registry-tp.service
$ sudo systemctl status sawtooth-validator.service
$ sudo systemctl status sawtooth-settings-tp.service
$ sudo systemctl status sawtooth-intkey-tp-python.service
```

Stop or Restart the Sawtooth Services

If you need to stop or restart the Sawtooth services for any reason, use the following commands:

Stop Sawtooth services:

```
$ sudo systemctl stop sawtooth-rest-api.service
$ sudo systemctl stop sawtooth-poet-validator-registry-tp.service
$ sudo systemctl stop sawtooth-validator.service
$ sudo systemctl stop sawtooth-settings-tp.service
$ sudo systemctl stop sawtooth-intkey-tp-python.service
```

Restart Sawtooth services:

```
$ sudo systemctl restart sawtooth-rest-api.service
$ sudo systemctl restart sawtooth-poet-validator-registry-tp.service
$ sudo systemctl restart sawtooth-validator.service
$ sudo systemctl restart sawtooth-settings-tp.service
$ sudo systemctl restart sawtooth-intkey-tp-python.service
```

CHAPTER SIX

API REFERENCES

6.1 SDK API Reference

6.1.1 Python

processor package

Submodules

processor.config module

`processor.config.`**`get_config_dir`**`()`

Returns the sawtooth configuration directory based on the SAWTOOTH_HOME environment variable (if set) or OS defaults.

`processor.config.`**`get_log_dir`**`()`

Returns the configured data directory.

`processor.config.`**`get_log_config`**`(`*filename=None*`)`

Returns the log config dictinary if it exists.

`processor.config.`**`get_processor_config`**`(`*filename=None*`)`

Returns the log config dictinary if it exists.

processor.context module

`class` `processor.context.`**`Context`**`(`*stream*, *context_id*`)`

Bases: `object`

Context provides an interface for getting, setting, and deleting validator state. All validator interactions by a handler should be through a Context instance.

Variables

- **`_stream`** (*`sawtooth.client.stream.Stream`*) – client grpc communication
- **`_context_id`** (*`str`*) – the context_id passed in from the validator

`get_state``(`*addresses*, *timeout=None*`)`

get_state queries the validator state for data at each of the addresses in the given list. The addresses that have been set are returned in a list.

Parameters

- **addresses** (*list*) – the addresses to fetch
- **timeout** – optional timeout, in seconds

Returns a list of Entries (address, data), for the addresses that have a value

Return type results (list)

Raises AuthorizationException

set_state (*entries*, *timeout=None*)

set_state requests that each address in the provided dictionary be set in validator state to its corresponding value. A list is returned containing the successfully set addresses.

Parameters

- **entries** (*dict*) – dictionary where addresses are the keys and data is the value.
- **timeout** – optional timeout, in seconds

Returns a list of addresses that were set

Return type addresses (list)

Raises AuthorizationException

delete_state (*addresses*, *timeout=None*)

delete_state requests that each of the provided addresses be unset in validator state. A list of successfully deleted addresses is returned.

Parameters

- **addresses** (*list*) – list of addresses to delete
- **timeout** – optional timeout, in seconds

Returns a list of addresses that were deleted

Return type addresses (list)

Raises AuthorizationException

add_receipt_data (*data*, *timeout=None*)

Add a blob to the execution result for this transaction.

Parameters **data** (*bytes*) – The data to add.

add_event (*event_type*, *attributes=None*, *data=None*, *timeout=None*)

Add a new event to the execution result for this transaction.

Parameters

- **event_type** (*str*) – This is used to subscribe to events. It should be globally unique and describe what, in general, has occured.
- **attributes** (*list of (str, str) tuples*) – Additional information about the event that is transparent to the validator. Attributes can be used by subscribers to filter the type of events they receive.
- **data** (*bytes*) – Additional information about the event that is opaque to the validator.

processor.core module

class processor.core.**TransactionProcessor** (*url*)

Bases: object

TransactionProcessor is a generic class for communicating with a validator and routing transaction processing requests to a registered handler. It uses ZMQ and channels to handle requests concurrently.

Parameters `url` (*string*) – The URL of the validator

zmq_id

add_handler(*handler*)
Adds a transaction family handler :param handler: the handler to be added :type handler: TransactionHandler

start()
Connects the transaction processor to a validator and starts listening for requests and routing them to an appropriate transaction handler.

stop()
Closes the connection between the TransactionProcessor and the validator.

processor.exceptions module

exception processor.exceptions.**InvalidTransaction**(*message*, *extended_data=None*)
Bases: `processor.exceptions._TpResponseError`

Raised for an Invalid Transaction.

exception processor.exceptions.**InternalError**(*message*, *extended_data=None*)
Bases: `processor.exceptions._TpResponseError`

Raised when an internal error occurs during transaction processing.

exception processor.exceptions.**AuthorizationException**
Bases: `Exception`

Raised when a authorization error occurs.

exception processor.exceptions.**LocalConfigurationError**
Bases: `Exception`

Raised when a log configuraiton error occurs.

processor.handler module

class processor.handler.**TransactionHandler**
Bases: `object`

TransactionHandler is the Abstract Base Class that defines the business logic for a new transaction family.

The family_name, family_versions, and namespaces properties are used by the processor to route processing requests to the handler.

family_name
family_name should return the name of the transaction family that this handler can process, e.g. "intkey"

family_versions
family_versions should return a list of versions this transaction family handler can process, e.g. ["1.0"]

namespaces
namespaces should return a list containing all the handler's namespaces, e.g. ["abcdef"]

apply (*transaction*, *context*)
Apply is the single method where all the business logic for a transaction family is defined. The method will be called by the transaction processor upon receiving a TpProcessRequest that the handler understands and will pass in the TpProcessRequest and an initialized instance of the Context type.

processor.log module

`processor.log.`**`create_console_handler`**(*verbose_level*)
Set up the console logging for a transaction processor. :param verbose_level: The log level that the console should print out :type verbose_level: int

`processor.log.`**`init_console_logging`**(*verbose_level=2*)
Set up the console logging for a transaction processor. :param verbose_level: The log level that the console should print out :type verbose_level: int

`processor.log.`**`log_configuration`**(*log_config=None*, *log_dir=None*, *name=None*)
Sets up the loggers for a transaction processor. :param log_config: A dictinary of log config options :type log_config: dict :param log_dir: The log directory's path :type log_dir: string :param name: The name of the expected logging file :type name: string

Module contents

The processor module defines:

1. A TransactionHandler interface to be used to create new transaction families.
2. A high-level, general purpose TransactionProcessor to which any number of handlers can be added.
3. A Context class used to abstract getting and setting addresses in global validator state.

sawtooth_signing package

Submodules

sawtooth_signing.core module

exception `sawtooth_signing.core.`**`NoSuchAlgorithmError`**
Bases: `Exception`

Thrown when trying to create an algorithm which does not exist.

exception `sawtooth_signing.core.`**`SigningError`**
Bases: `Exception`

Thrown when an error occurs during the signing process.

exception `sawtooth_signing.core.`**`ParseError`**
Bases: `Exception`

Thrown when an error occurs during deserialization of a Private or Public key from various formats.

class `sawtooth_signing.core.`**`PrivateKey`**
Bases: `object`

A private key instance.

The underlying content is dependent on implementation.

get_algorithm_name()
Returns the algorithm name used for this private key.

as_hex()
Return the private key encoded as a hex string.

as_bytes()
Return the private key bytes.

class sawtooth_signing.core.**PublicKey**
Bases: `object`

A public key instance.

The underlying content is dependent on implementation.

get_algorithm_name()
Returns the algorithm name used for this public key.

as_hex()
Return the public key encoded as a hex string.

as_bytes()
Return the public key bytes.

class sawtooth_signing.core.**Context**
Bases: `object`

A context for a cryptographic signing algorithm.

get_algorithm_name()
Returns the algorithm name.

sign(*message*, *private_key*)
Sign a message

Given a private key for this algorithm, sign the given message bytes and return a hex-encoded string of the resulting signature.

Parameters

- **message** (*bytes*) – the message bytes
- **private_key** (*PrivateKey*) – the private key

Returns The signature in a hex-encoded string

Raises *SigningError* – if any error occurs during the signing process

verify(*signature*, *message*, *public_key*)
Verifies that a signature of a message was produced with the associated public key.

Parameters

- **signature** (*str*) – the hex-encoded signature
- **message** (*bytes*) – the message bytes
- **public_key** (*PublicKey*) – the public key to use for verification

Returns True if the public key is associated with the signature for that method, False otherwise

Return type boolean

new_random_private_key()
Generates a new random PrivateKey using this context.

Returns a random private key

Return type (*PrivateKey*)

get_public_key(*private_key*)

Produce a public key for the given private key.

Parameters **private_key** (*PrivateKey*) – a private key

Returns (*PublicKey*) the public key for the given private key

sawtooth_signing.secp256k1 module

class sawtooth_signing.secp256k1.**Secp256k1PrivateKey**(*secp256k1_private_key*)

Bases: *sawtooth_signing.core.PrivateKey*

get_algorithm_name()

Returns the algorithm name used for this private key.

as_hex()

Return the private key encoded as a hex string.

as_bytes()

Return the private key bytes.

secp256k1_private_key

static from_hex()

static new_random()

class sawtooth_signing.secp256k1.**Secp256k1PublicKey**(*secp256k1_public_key*)

Bases: *sawtooth_signing.core.PublicKey*

secp256k1_public_key

get_algorithm_name()

Returns the algorithm name used for this public key.

as_hex()

Return the public key encoded as a hex string.

as_bytes()

Return the public key bytes.

static from_hex()

class sawtooth_signing.secp256k1.**Secp256k1Context**

Bases: *sawtooth_signing.core.Context*

get_algorithm_name()

Returns the algorithm name.

sign(*message*, *private_key*)

Sign a message

Given a private key for this algorithm, sign the given message bytes and return a hex-encoded string of the resulting signature.

Parameters

- **message** (*bytes*) – the message bytes
- **private_key** (PrivateKey) – the private key

Returns The signature in a hex-encoded string

Raises `SigningError` – if any error occurs during the signing process

verify (*signature*, *message*, *public_key*)

Verifies that a signature of a message was produced with the associated public key.

Parameters

- **signature** (*str*) – the hex-encoded signature
- **message** (*bytes*) – the message bytes
- **public_key** (`PublicKey`) – the public key to use for verification

Returns True if the public key is associated with the signature for that method, False otherwise

Return type boolean

new_random_private_key ()

Generates a new random PrivateKey using this context.

Returns a random private key

Return type (`PrivateKey`)

get_public_key (*private_key*)

Produce a public key for the given private key.

Parameters **private_key** (`PrivateKey`) – a private key

Returns (`PublicKey`) the public key for the given private key

Module contents

class sawtooth_signing.**Signer** (*context*, *private_key*)

Bases: `object`

A convenient wrapper of Context and PrivateKey

sign (*message*)

Signs the given message

Parameters **message** (*bytes*) – the message bytes

Returns The signature in a hex-encoded string

Raises `SigningError` – if any error occurs during the signing process

get_public_key ()

Return the public key for this Signer instance.

class sawtooth_signing.**CryptoFactory** (*context*)

Bases: `object`

Factory for generating Signers.

context

Return the context that backs this factory instance

new_signer (*private_key*)

Create a new signer for the given private key.

Parameters **private_key** (`PrivateKey`) – a private key

Returns a signer instance

Return type (*Signer*)

`sawtooth_signing.`**`create_context`**(*algorithm_name*)

Returns an algorithm instance by name.

Parameters **`algorithm_name`** (*`str`*) – the algorithm name

Returns a context instance for the given algorithm

Return type (`Context`)

Raises NoSuchAlgorithmError if the algorithm is unknown

6.1.2 Go

- Transaction Processor
- Signing

6.1.3 Javascript

- Transaction Processor
- Signing

6.1.4 Rust

- Transaction Processor
- Signing

6.2 REST API Reference

6.2.1 Endpoint Specifications

These specifications document the available Sawtooth REST API endpoints. They are generated from the official OpenAPI formatted YAML specification, which can be found on the Sawtooth GitHub.

`POST /batches`

Sends a BatchList to the validator

Accepts a protobuf formatted *BatchList* as an octet-stream binary file and submits it to the validator to be committed.

The API will return immediately with a status of *202*. There will be no *data* object, only a *link* to a */batch_statuses* endpoint to be polled to check the status of submitted batches.

Status Codes

- 202 Accepted – Batches submitted for validation, but not yet committed
- 400 Bad Request – Request was malformed
- 429 Too Many Requests – Too many requests have been made to process batches
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /batches

Fetches a list of batches

Fetches a paginated list of batches from the validator.

Query Parameters

- **head** (*string*) – Index or id of head block
- **start** (*string*) – Id to start paging (inclusive)
- **limit** (*integer*) – Number of items to return
- **reverse** (*string*) – If the list should be reversed

Status Codes

- 200 OK – Successfully retrieved batches
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /batches/{batch_id}

Fetches a particular batch

Parameters

- **batch_id** (*string*) – Batch id

Status Codes

- 200 OK – Successfully retrieved batch
- 400 Bad Request – Request was malformed
- 404 Not Found – Address or id did not match any resource
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /batch_statuses

Fetches the committed statuses for a set of batches

Fetches an array of objects with a status and id for each batch requested. There are four possible statuses with string values *'COMMITTED'*, *'INVALID'*, *'PENDING'*, and *'UNKNOWN'*.

The batch(es) you want to check can be specified using the *id* filter parameter. If a *wait* time is specified in the URL, the API will wait to respond until all batches are committed, or the time in seconds has elapsed. If the value of *wait* is not set (i.e., *?wait&id=…*), or it is set to any non-integer value other than *false*, the wait time will be just under the API's specified timeout (usually 300).

Note that because this route does not return full resources, the response will not be paginated, and there will be no *head* or *paging* properties.

Query Parameters

- **id** (*string*) – A comma-separated list of batch ids
- **wait** (*integer*) – A time in seconds to wait for commit

Status Codes

- 200 OK – Successfully retrieved statuses
- 400 Bad Request – Request was malformed

- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

POST /batch_statuses

Fetches the committed statuses for a set of batches

Identical to *GET /batch_statuses*, but takes ids of batches as a JSON formatted POST body rather than a query parameter. This allows for many more batches to be checked and should be used for more than 15 ids.

Note that because query information is not encoded in the URL, no *link* will be returned with this query.

Query Parameters

- **wait** (*integer*) – A time in seconds to wait for commit

Status Codes

- 200 OK – Successfully retrieved statuses
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /state

Fetches the data for the current state

Fetches a paginated list of entries for the current state, or relative to a particular head block. Using the *address* filter parameter will narrow the list to any entries that have an address beginning with the characters specified.

Query Parameters

- **head** (*string*) – Index or id of head block
- **address** (*string*) – A partial address to filter leaves by
- **start** (*string*) – Id to start paging (inclusive)
- **limit** (*integer*) – Number of items to return
- **reverse** (*string*) – If the list should be reversed

Status Codes

- 200 OK – Successfully retrieved state data
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /state/{address}

Fetches a particular leaf from the current state

Parameters

- **address** (*string*) – Radix address of a leaf

Query Parameters

- **head** (*string*) – Index or id of head block

Status Codes

- 200 OK – Successfully fetched leaves
- 400 Bad Request – Request was malformed

- 404 Not Found – Address or id did not match any resource
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

`GET /blocks`
Fetches a list of blocks

Fetches a paginated list of blocks from the validator.

Query Parameters

- **`head`** (*`string`*) – Index or id of head block
- **`start`** (*`string`*) – Id to start paging (inclusive)
- **`limit`** (*`integer`*) – Number of items to return
- **`reverse`** (*`string`*) – If the list should be reversed

Status Codes

- 200 OK – Successfully retrieved blocks
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

`GET /blocks/{block_id}`
Fetches a particular block

Parameters

- **`block_id`** (*`string`*) – Block id

Status Codes

- 200 OK – Successfully retrieved block
- 400 Bad Request – Request was malformed
- 404 Not Found – Address or id did not match any resource
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

`GET /transactions`
Fetches a list of transactions

Fetches a paginated list of transactions from the validator.

Query Parameters

- **`head`** (*`string`*) – Index or id of head block
- **`start`** (*`string`*) – Id to start paging (inclusive)
- **`limit`** (*`integer`*) – Number of items to return
- **`reverse`** (*`string`*) – If the list should be reversed

Status Codes

- 200 OK – Successfully retrieved transactions
- 400 Bad Request – Request was malformed

- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /transactions/{transaction_id}
Fetches a particular transaction

Parameters

- **transaction_id** (*string*) – Transaction id

Status Codes

- 200 OK – Successfully retrieved transaction
- 400 Bad Request – Request was malformed
- 404 Not Found – Address or id did not match any resource
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /receipts
Fetches the receipts for a set of transactions

Fetches an array of objects for each receipt requested.

The receipt(s) you want to retrieve can be specified using the *id* filter parameter, where *id* refers to the transaction id of the transaction the receipt is associated with.

Query Parameters

- **id** (*string*) – A comma-separated list of transaction ids

Status Codes

- 200 OK – Successfully retrieved transaction receipts
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

POST /receipts
Fetches the receipts for a set of transactions

Identical to *GET /receipts*, but takes ids of transactions as a JSON formatted POST body rather than a query parameter. This allows for many more receipts to be fetched and should be used with more than 15 ids.

Note that because query information is not encoded in the URL, no *link* will be returned with this request.

Query Parameters

- **wait** (*integer*) – A time in seconds to wait for commit

Status Codes

- 200 OK – Successfully retrieved transaction receipts
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

GET /peers
Fetches the endpoints of the authorized peers of the validator

Status Codes

- 200 OK – Successfully retrieved peers
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

`GET /status`

Fetches information pertaining to the status of the validator

Status Codes

- 200 OK – Successfully retrieved status
- 400 Bad Request – Request was malformed
- 500 Internal Server Error – Something went wrong within the validator
- 503 Service Unavailable – API is unable to reach the validator

6.2.2 Error Responses

When the REST API encounters a problem, or receives notification that the validator has encountered a problem, it will notify clients with both an appropriate HTTP status code and a more detailed JSON response.

HTTP Status Codes

Code	Title	Description
400	Bad Request	The request was malformed in some way, and will need to be modified before resubmitting. The accompanying JSON response will have more details.
404	Not Found	The request was well formed, but the specified identifier did not correspond to any resource in the validator. Returned by endpoints which fetch a single resource. Endpoints which return lists of resources will simply return an empty list.
500	Internal Server Error	Something is broken internally in the REST API or the validator. This may be a bug; if it is reproducible, the bug should be reported.
503	Service Unavailable	The REST API is unable to communicate with the validator. It may be down. You should try your request later.

JSON Response

In the case of an error, rather than a *data* property, the JSON response will include a single *error* property with three values:

- *code* (integer) - a machine readable error code
- *title* (string) - a short headline for the error
- *message* (string) - a longer, human-readable description of what went wrong

Note: While the title and message may change in the future, the error code will **not** change; it is fixed and will always refer to this particular problem.

Example JSON Response

```
{
  "error": {
    "code": 30,
    "title": "Submitted Batches Invalid",
    "message": "The submitted BatchList is invalid. It was poorly formed or has an␣
↪invalid signature."
  }
}
```

Error Codes and Descriptions

Code	Title	Description
10	Unknown Validator Error	An unknown error occurred with the validator while processing the request. This may be a bug; if it is reproducible, the bug should be reported.
15	Validator Not Ready	The validator has no genesis block, and so cannot be queried. Wait for genesis to be completed and resubmit. If you are running the validator, ensure it was set up properly.
17	Validator Timed Out	The request timed out while waiting for a response from the validator. It may not be running, or may have encountered an internal error. The request may not have been processed.
18	Validator Disconnected	The validator sent a disconnect signal while processing the response, and is no longer available. Try your request again later.
20	Invalid Validator Response	The validator sent back a response which was not serialized properly and could not be decoded. There may be a problem with the validator.
21	Invalid Resource Header	The validator sent back a resource with a header that could not be decoded. There may be a problem with the validator, or the data may have been corrupted.
27	Unable to Fetch Statuses	The validator should always return some status for every batch requested. An unknown error caused statuses to be missing, and should be reported.
30	Submitted Batches Invalid	The submitted BatchList failed initial validation by the validator. It may have a bad signature or be poorly formed.
31	Unable to Accept Batches	The validator cannot currently accept more batches due to a full queue. Please submit your request again.
34	No Batches Submitted	The BatchList Protobuf submitted was empty and contained no batches. All submissions to the validator must include at least one batch.
35	Protobuf Not Decodable	The REST API was unable to decode the submitted Protobuf binary. It is poorly formed, and has not been submitted to the validator.
42	Wrong Content Type (submit batches)	POST requests to submit a BatchList must have a ‘Content-Type’ header of ‘application/octet-stream’.
43	Wrong Content Type (fetch statuses)	If using a POST request to fetch batch statuses, the ‘Content-Type’ header must be ‘application/json’.
46	Bad Status Request	The body of the POST request to fetch batch statuses was poorly formed. It must be a JSON formatted array of string-formatted batch ids, with at least one id.
50	Head Not Found	A ‘head’ query parameter was used, but the block id specified does not correspond to any block in the validator.
53	Invalid Count Query	The ‘count’ query parameter must be a positive, non-zero integer.
54	Invalid Paging Query	The validator rejected the paging request submitted. One or more of the ‘min’, ‘max’, or ‘count’ query parameters were invalid or out of range.
57	Invalid Sort Query	The validator rejected the sort request submitted. Most likely one of the keys specified was not found in the resources sorted.
60	Invalid Resource Id	A submitted block, batch, or transaction id was invalid. All such resources are identified by 128 character hex-strings.
62	Invalid State Address	The state address submitted was invalid. Returned when attempting to fetch a particular “leaf” from the state tree. When fetching specific state data, the full 70-character address must be used.
66	Id Query Invalid or Missing	If using a GET request to fetch batch statuses, an ‘id’ query parameter must be specified, with a comma-separated list of at least one batch id.
70	Block Not Found	There is no block with the id specified in the blockchain.
71	Batch Not Found	There is no batch with the id specified in the blockchain.
72	Transaction Not Found	There is no transaction with the id specified in the blockchain.
75	State Not Found	There is no state data at the address specified.
80	Transaction Receipt Not Found	There is no transaction receipt for the transaction id specified in the receipt store.
81	Wrong Content Type	Requests for transaction receipts sent as a POST must have a ‘Content-Type’ header of ‘application/json’.
82	Bad Receipts Re-	Requests for transaction receipts sent as a POST must have a JSON formatted body

6.2.3 State Delta Subscriptions via Web Sockets

As transactions are committed to the blockchain, an app developer may be interested in receiving events related to the changes in state that result. These events, *StateDeltaEvents*, include information about the advance of the blockchain, as well as state changes that can be limited to specific address spaces in the global state.

An application can subscribe to receive these events via a web socket, provided by the REST API component. For example, a single-page JavaScript application may open a web socket connection and subscribe to a particular transaction family's state values, using the incoming events to re-render portions of the display.

Note: All examples here are written in JavaScript, and assumes the Sawtooth REST API is reachable at *localhost*.

Opening a Web Socket

The application developer must first open a web socket. This is accomplished by using standard means. In the case of in-browser JavaScript:

```
let ws = new WebSocket('ws:localhost:8008/subscriptions')
```

If the REST API is running, it should trigger an event on the web socket's *onopen* handler.

Subscribing to State Changes

In order to subscribe to an address space in the global state, first a message needs to be sent on the socket with the list of prefixes. It is a best-practice to send this message as part of the web socket's *onopen* handler.

In the following example, we'll subscribe to changes in the XO family:

```
ws.onopen = () => {
  ws.send(JSON.stringify({
    'action': 'subscribe',
    'address_prefixes': ['5b7349']
  }))
}
```

This message will begin the subscription of events as of the current block. If you are interested in the state prior to the point of subscription, you should fetch the values of state via the REST API's */state* endpoint.

Subscriptions may be changed by sending a subscribe message at later time while the websocket is open. It is up to the client to maintain the list of address prefixes of interest. Any subsequent subscriptions will overwrite this list.

Events

Once subscribed, events will be received via the web socket's *onmessage* handler. The event data is a JSON string, which looks like the following:

```
{
  "block_num": 8,
  "block_id": "ab7cbc7a...",
  "previous_block_id": "d4b46c1c...",
  "state_changes": [
    {
      "type": "SET",
```

(continues on next page)

(continued from previous page)

```
      "value": "oWZQdmxqcmsZU4w"=,
      "address": "1cf126613a..."
    },
    ...
  ]
}
```

There is an entry in the *state_changes* array for each address that matches the *address_prefixes* provided during the subscribe action. The type is either "SET" or "DELETE". In the case of "SET" the value is base-64 encoded (like the */state* endpoint's response). In the case of "DELETE", only the address is provided. If you are using a transaction family that supports deletes, you'll need to keep track of values via address, as well.

Missed Events

In the case where you have missed an event, a request can be sent via the web socket for a particular block's changes. You can use the *previous_block_id* from the current event to request the previous block's events, for example. Send the following message:

```
ws.send(JSON.stringify({
  'action': 'get_block_deltas',
  'block_id': 'd4b46c1c...',
  'address_prefixes': ['5b7349']
}))
```

The event will be returned in the same manner as any other event, so it is recommended that you push the events on to a stack before processing them.

If the block id does not exist, the following error will be returned:

```
{
  "error": "Must specify a block id"
}
```

Unsubscribing

To unsubscribe, you can either close the web socket, or if you want to unsubscribe temporarily, you can send an unsubscribe action:

```
ws.send(JSON.stringify({
  'action': 'unsubscribe'
}))
```

Errors and Warnings

An open, subscribed web socket may receive the following errors and warnings:

- the validator is unavailable
- an unknown action was requested

If the validator is unavailable to the REST API process, a warning will be sent in lieu of a state delta event:

```
{
  "warning": "Validator unavailable"
}
```

If an unrecognized action is sent on to the server via the websocket, an error message will be sent back:

```
{
  "error": "Unknown action \"bad_action\""
}
```

CHAPTER

SEVEN

CLI COMMAND REFERENCE

The Sawtooth command-line interface (CLI) provides a set of commands to interact with Sawtooth services.

This chapter shows the available options and arguments for each command and subcommand. The synopsis for each command shows its parameters and their usage.

- Optional parameters are shown in square brackets
- Choices are shown in curly braces.
- User-supplied values are shown in angle brackets.

This usage information is also available on the command line by using the -h or --help option.

7.1 sawtooth

The sawtooth command is the usual way to interact with validators or validator networks.

This command has a multi-level structure. It starts with the base call to sawtooth. Next is a top-level subcommand such as block or state. Each top-level subcommand has additional subcommands that specify the operation to perform, such as list or create. The subcommands have options and arguments that control their behavior. For example:

```
$ sawtooth state list --format csv
```

```
usage: sawtooth [-h] [-v] [-V]
                {batch,block,identity,keygen,peer,status,settings,state,transaction}
                ...

Provides subcommands to configure, manage, and use Sawtooth components.

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information

subcommands:
  {batch,block,identity,keygen,peer,status,settings,state,transaction}
    batch               Displays information about batches and submit new
                        batches
    block               Displays information on blocks in the current
                        blockchain
    identity            Works with optional roles, policies, and permissions
    keygen              Creates user signing keys
```

(continues on next page)

(continued from previous page)

```
    peer                Displays information about validator peers
    status              Displays information about validator status
    settings            Displays on-chain settings
    state               Displays information on the entries in state
    transaction         Shows information on transactions in the current chain
```

7.1.1 sawtooth batch

The `sawtooth batch` subcommands display information about the Batches in the current blockchain and submit Batches to the validator via the REST API. A Batch is a group of interdependent transactions that is the atomic unit of change in Sawtooth. For more information, see “Transactions and Batches!”

```
usage: sawtooth batch [-h] {list,show,status,submit} ...

Provides subcommands to display Batch information and submit Batches to the
validator via the REST API.

optional arguments:
  -h, --help            show this help message and exit

subcommands:
  {list,show,status,submit}
```

7.1.2 sawtooth batch list

The `sawtooth batch list` subcommand queries the specified Sawtooth REST API (default: `http://localhost:8008`) for a list of Batches in the current blockchain. It returns the id of each Batch, the public key of each signer, and the number of transactions in each Batch.

By default, this information is displayed as a white-space delimited table intended for display, but other plain-text formats (CSV, JSON, and YAML) are available and can be piped into a file for further processing.

```
usage: sawtooth batch list [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                           [-F {csv,json,yaml,default}]

Displays all information about all committed Batches for the specified validator,␣
↪including the Batch id, public keys of all signers, and number of transactions in␣
↪each Batch.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -F {csv,json,yaml,default}, --format {csv,json,yaml,default}
                        choose the output format
```

7.1.3 sawtooth batch show

The `sawtooth batch show` subcommand queries the Sawtooth REST API for a specific batch in the current blockchain. It returns complete information for this batch in either YAML (default) or JSON format. Use the `--key` option to narrow the returned information to just the value of a single key, either from the batch or its header.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth batch show [-h] [--url URL] [-u USERNAME[:PASSWORD]] [-k KEY]
                           [-F {yaml,json}]
                           batch_id

Displays information for the specified Batch.

positional arguments:
  batch_id              id (header_signature) of the batch

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -k KEY, --key KEY     show a single property from the block or header
  -F {yaml,json}, --format {yaml,json}
                        choose the output format (default: yaml)
```

7.1.4 sawtooth batch status

The `sawtooth batch status` subcommand queries the Sawtooth REST API for the committed status of one or more batches, which are specified as a list of comma-separated Batch ids. The output is in either YAML (default) or JSON format, and includes the ids of any invalid transactions with an error message explaining why they are invalid. The `--wait` option indicates that results should not be returned until processing is complete, with an optional timeout value specified in seconds.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth batch status [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                             [--wait [WAIT]] [-F {yaml,json}]
                             batch_ids

Displays the status of the specified Batch id or ids.

positional arguments:
  batch_ids             single batch id or comma-separated list of batch ids

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  --wait [WAIT]         set time, in seconds, to wait for commit
  -F {yaml,json}, --format {yaml,json}
                        choose the output format (default: yaml)
```

7.1.5 sawtooth batch submit

The `sawtooth batch submit` subcommand sends one or more Batches to the Sawtooth REST API to be submitted to the validator. The input is a binary file with a binary-encoded `BatchList` protobuf, which can contain one or more batches with any number of transactions. The `--wait` option indicates that results should not be returned until processing is complete, with an optional timeout specified in seconds.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth batch submit [-h] [--url URL] [-u USERNAME[:PASSWORD]] [-v]
                             [-V] [--wait [WAIT]] [-f FILENAME]
                             [--batch-size-limit BATCH_SIZE_LIMIT]

Sends Batches to the REST API to be submitted to the validator. The input must
be a binary file containing a binary-encoded BatchList of one or more batches
with any number of transactions.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --wait [WAIT]         set time, in seconds, to wait for batches to commit
  -f FILENAME, --filename FILENAME
                        specify location of input file
  --batch-size-limit BATCH_SIZE_LIMIT
                        set maximum batch size; batches are split for
                        processing if they exceed this size
```

7.1.6 sawtooth block

The `sawtooth block` subcommands display information about the blocks in the current blockchain.

```
usage: sawtooth block [-h] {list,show} ...

Provides subcommands to display information about the blocks in the current
blockchain.

optional arguments:
  -h, --help   show this help message and exit

subcommands:
  {list,show}
    list       Displays information for all blocks on the current blockchain
    show       Displays information about the specified block on the current
               blockchain
```

7.1.7 sawtooth block list

The `sawtooth block list` subcommand queries the Sawtooth REST API (default: `http://localhost:8008`) for a list of blocks in the current chain. Using the `--count` option, the number of blocks returned can be configured. It returns the id and number of each block, the public key of each signer, and the number of transactions and batches in each.

By default, this information is displayed as a white-space delimited table intended for display, but other plain-text formats (CSV, JSON, and YAML) are available and can be piped into a file for further processing.

```
usage: sawtooth block list [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                           [-F {csv,json,yaml,default}] [-n COUNT]

Displays information for all blocks on the current blockchain, including the block id␣
↪and number, public keys all of allsigners, and number of transactions and batches.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -F {csv,json,yaml,default}, --format {csv,json,yaml,default}
                        choose the output format
  -n COUNT, --count COUNT
                        the number of blocks to list
```

7.1.8 sawtooth block show

The `sawtooth block show` subcommand queries the Sawtooth REST API for a specific block in the current blockchain. It returns complete information for this block in either YAML (default) or JSON format. Using the `--key` option, it is possible to narrow the returned information to just the value of a single key, either from the block, or its header.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth block show [-h] [--url URL] [-u USERNAME[:PASSWORD]] [-k KEY]
                           [-F {yaml,json}]
                           block_id

Displays information about the specified block on the current blockchain.

positional arguments:
  block_id              id (header_signature) of the block

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -k KEY, --key KEY     show a single property from the block or header
  -F {yaml,json}, --format {yaml,json}
                        choose the output format (default: yaml)
```

7.1.9 sawtooth identity

Sawtooth supports an identity system that provides an extensible role- and policy-based system for defining permissions in a way which can be used by other pieces of the architecture. This includes the existing permissioning components for transactor key and validator key; in the future, this feature may also be used by transaction family implementations. The `sawtooth identity` subcommands can be used to view the current roles and policy set in state, create new roles, and new policies.

Note that only the public keys stored in the setting sawtooth.identity.allowed_keys are allowed to submit identity transactions. Use the `sawset` commands to change this setting.

```
usage: sawtooth identity [-h] {policy,role} ...

Provides subcommands to work with roles and policies.

optional arguments:
  -h, --help     show this help message and exit

subcommands:
  {policy,role}
    policy       Provides subcommands to display existing policies and create
                 new policies
    role         Provides subcommands to display existing roles and create new
                 roles
```

7.1.10 sawtooth identity policy

The `sawtooth identity policy` subcommands are used to display the current policies stored in state and to create new policies.

```
usage: sawtooth identity policy [-h] {create,list} ...

Provides subcommands to list the current policies stored in state and to
create new policies.

optional arguments:
  -h, --help     show this help message and exit

policy:
  {create,list}
    create       Creates batches of sawtooth-identity transactions for setting
                 a policy
    list         Lists the current policies
```

7.1.11 sawtooth identity policy create

The `sawtooth identity policy create` subcommand creates a new policy that can then be set to a role. The policy should contain at least one “rule” (`PERMIT_KEY` or `DENY_KEY`). Note that all policies have an assumed last rule to deny all. This subcommand can also be used to change the policy that is already set to a role without having to also reset the role.

```
usage: sawtooth identity policy create [-h] [-k KEY] [-o OUTPUT | --url URL]
                                       [--wait WAIT]
                                       name rule [rule ...]

Creates a policy that can be set to a role or changes a policy without
resetting the role.

```

(continues on next page)

(continued from previous page)

```
positional arguments:
  name                  name of the new policy
  rule                  rule with the format "PERMIT_KEY <key>" or "DENY_KEY
                        <key> (multiple "rule" arguments can be specified)

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     specify the signing key for the resulting batches
  -o OUTPUT, --output OUTPUT
                        specify the output filename for the resulting batches
  --url URL             identify the URL of a validator's REST API
  --wait WAIT           set time, in seconds, to wait for the policy to commit
                        when submitting to the REST API.
```

7.1.12 sawtooth identity policy list

The `sawtooth identity policy list` subcommand lists the policies that are currently set in state. This list can be used to figure out which policy name should be set for a new role.

```
usage: sawtooth identity policy list [-h] [--url URL]
                                     [--format {default,csv,json,yaml}]

Lists the policies that are currently set in state.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of a validator's REST API
  --format {default,csv,json,yaml}
                        choose the output format
```

7.1.13 sawtooth identity role

The `sawtooth identity role` subcommands are used to list the current roles stored in state and to create new roles.

```
usage: sawtooth identity role [-h] {create,list} ...

Provides subcommands to list the current roles stored in state and to create
new roles.

optional arguments:
  -h, --help     show this help message and exit

role:
  {create,list}
    create       Creates a new role that can be used to enforce permissions
    list         Lists the current keys and values of roles
```

7.1.14 sawtooth identity role create

The `sawtooth identity role create` subcommand creates a new role that can be used to enforce permissions. The policy argument identifies the policy that the role is restricted to. This policy must already exist and be stored in state. Use `sawtooth identity policy list` to display the existing policies. The role name should reference an action that can be taken on the network. For example, the role named `transactor.transaction_signer` controls who is allowed to sign transactions.

```
usage: sawtooth identity role create [-h] [-k KEY] [--wait WAIT]
                                     [-o OUTPUT | --url URL]
                                     name policy

Creates a new role that can be used to enforce permissions.

positional arguments:
  name                  name of the role
  policy                identify policy that role will be restricted to

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     specify the signing key for the resulting batches
  --wait WAIT           set time, in seconds, to wait for a role to commit
                        when submitting to the REST API.
  -o OUTPUT, --output OUTPUT
                        specify the output filename for the resulting batches
  --url URL             the URL of a validator's REST API
```

7.1.15 sawtooth identity role list

The `sawtooth identity role list` subcommand displays the roles that are currently set in state. This list can be used to determine which permissions are being enforced on the network. The output includes which policy the roles are set to.

By default, this information is displayed as a white-space delimited table intended for display, but other plain-text formats (CSV, JSON, and YAML) are available and can be piped into a file for further processing.

```
usage: sawtooth identity role list [-h] [--url URL]
                                   [--format {default,csv,json,yaml}]

Displays the roles that are currently set in state.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of a validator's REST API
  --format {default,csv,json,yaml}
                        choose the output format
```

7.1.16 sawtooth keygen

The `sawtooth keygen` subcommand generates a private key file and a public key file so that users can sign Sawtooth transactions and batches. These files are stored in the `<key-dir>` directory in `<key_name>.priv` and `<key_dir>/<key_name>.pub`. By default, `<key_dir>` is `~/.sawtooth` and `<key_name>` is `$USER`.

```
usage: sawtooth keygen [-h] [-v] [-V] [--key-dir KEY_DIR] [--force] [-q]
                       [key_name]

Generates keys with which the user can sign transactions and batches.

```

(continues on next page)

(continued from previous page)

```
positional arguments:
  key_name           specify the name of the key to create

optional arguments:
  -h, --help         show this help message and exit
  -v, --verbose      enable more verbose output
  -V, --version      display version information
  --key-dir KEY_DIR  specify the directory for the key files
  --force            overwrite files if they exist
  -q, --quiet        do not display output

The private and public key files are stored in <key-dir>/<key-name>.priv and
<key-dir>/<key-name>.pub. <key-dir> defaults to ~/.sawtooth and <key-name>
defaults to $USER.
```

7.1.17 sawtooth peer

The `sawtooth peer` subcommand displays the addresses of a specified validator's peers.

```
usage: sawtooth peer [-h] {list} ...

Provides a subcommand to list a validator's peers

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  {list}
```

7.1.18 sawtooth peer list

The `sawtooth peer list` subcommand displays the addresses of a specified validator's peers.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth peer list [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                          [-F {csv,json,yaml,default}]

Displays the addresses of the validators with which a specified validator is
peered.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -F {csv,json,yaml,default}, --format {csv,json,yaml,default}
                        choose the output format
```

7.1.19 sawtooth settings

The `sawtooth settings` subcommand displays the values of currently active on-chain settings.

```
usage: sawtooth settings [-h] {list} ...

Displays the values of currently active on-chain settings.

optional arguments:
  -h, --help  show this help message and exit

settings:
  {list}
    list      Lists the current keys and values of on-chain settings
```

7.1.20 sawtooth settings list

The `sawtooth settings list` subcommand displays the current keys and values of on-chain settings.

```
usage: sawtooth settings list [-h] [--url URL] [--filter FILTER]
                              [--format {default,csv,json,yaml}]

List the current keys and values of on-chain settings. The content can be
exported to various formats for external consumption.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of a validator's REST API
  --filter FILTER       filters keys that begin with this value
  --format {default,csv,json,yaml}
                        choose the output format
```

7.1.21 sawtooth state

The `sawtooth state` subcommands display information about the entries in the current blockchain state.

```
usage: sawtooth state [-h] {list,show} ...

Provides subcommands to display information about the state entries in the
current blockchain state.

optional arguments:
  -h, --help   show this help message and exit

subcommands:
  {list,show}
```

7.1.22 sawtooth state list

The `sawtooth state list` subcommand queries the Sawtooth REST API for a list of all state entries in the current blockchain state. This subcommand returns the address of each entry, its size in bytes, and the byte-encoded data it contains. It also returns the head block for which this data is valid.

To control the state that is returned, use the `subtree` argument to specify an address prefix as a filter or a block id to use as the chain head.

By default, this information is displayed as a white-space delimited table intended for display, but other plain-text formats (CSV, JSON, and YAML) are available and can be piped into a file for further processing.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth state list [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                           [-F {csv,json,yaml,default}] [--head HEAD]
                           [subtree]

Lists all state entries in the current blockchain.

positional arguments:
  subtree               address of a subtree to filter the list by

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -F {csv,json,yaml,default}, --format {csv,json,yaml,default}
                        choose the output format
  --head HEAD           specify the id of the block to set as the chain head
```

7.1.23 sawtooth state show

The `sawtooth state show` subcommand queries the Sawtooth REST API for a specific state entry (address) in the current blockchain state. It returns the data stored at this state address and the id of the chain head for which this data is valid. This data is byte-encoded per the logic of the transaction family that created it, and must be decoded using that same logic.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

By default, the peers are displayed as a CSV string, but other plain-text formats (JSON, and YAML) are available and can be piped into a file for further processing.

```
usage: sawtooth state show [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                           [--head HEAD]
                           address

Displays information for the specified state address in the current blockchain.

positional arguments:
  address               address of the leaf

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  --head HEAD           specify the id of the block to set as the chain head
```

7.1.24 sawtooth status

The `sawtooth status` subcommands display information related to a validator's status.

```
usage: sawtooth status [-h] {show} ...

Provides a subcommand to show a validator's status

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  {show}
```

7.1.25 sawtooth status show

The `sawtooth status` subcommand displays information related to a validator's current status, including its public network endpoint and its peers.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth status show [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                            [-F {csv,json,yaml,default}]

Displays information about the status of a validator.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -F {csv,json,yaml,default}, --format {csv,json,yaml,default}
                        choose the output format
```

7.1.26 sawtooth transaction

The `sawtooth transaction` subcommands display information about the transactions in the current blockchain.

```
usage: sawtooth transaction [-h] {list,show} ...

Provides subcommands to display information about the transactions in the
current blockchain.

optional arguments:
  -h, --help   show this help message and exit

subcommands:
  {list,show}
```

7.1.27 sawtooth transaction list

The `sawtooth transaction list` subcommand queries the Sawtooth REST API (default: `http://localhost:8008`) for a list of transactions in the current blockchain. It returns the id of each transaction, its family and version, the size of its payload, and the data in the payload itself.

By default, this information is displayed as a white-space delimited table intended for display, but other plain-text formats (CSV, JSON, and YAML) are available and can be piped into a file for further processing.

```
usage: sawtooth transaction list [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                                 [-F {csv,json,yaml,default}]

Lists all transactions in the current blockchain.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -F {csv,json,yaml,default}, --format {csv,json,yaml,default}
                        choose the output format
```

7.1.28 sawtooth transaction show

The `sawtooth transaction show` subcommand queries the Sawtooth REST API for a specific transaction in the current blockchain. It returns complete information for this transaction in either YAML (default) or JSON format. Use the `--key` option to narrow the returned information to just the value of a single key, either from the transaction or its header.

This subcommand requires the URL of the REST API (default: `http://localhost:8008`), and can specify a *username*:*password* combination when the REST API is behind a Basic Auth proxy.

```
usage: sawtooth transaction show [-h] [--url URL] [-u USERNAME[:PASSWORD]]
                                 [-k KEY] [-F {yaml,json}]
                                 transaction_id

Displays information for the specified transaction.

positional arguments:
  transaction_id        id (header_signature) of the transaction

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of the validator's REST API (default:
                        http://localhost:8008)
  -u USERNAME[:PASSWORD], --user USERNAME[:PASSWORD]
                        specify the user to authorize request
  -k KEY, --key KEY     show a single property from the block or header
  -F {yaml,json}, --format {yaml,json}
                        choose the output format (default: yaml)
```

7.2 sawtooth-validator

The `sawtooth-validator` command controls the behavior of the validator.

A validator is the component ultimately responsible for validating batches of transactions, combining them into blocks, maintaining consensus with the network, and coordinating communication between clients, other validators, and transaction processors. Much of the actual validation is delegated to other components, such as transaction processors and the active consensus module.

Note the following options, which provide several ways to configure the validator.

- Use the `--peering` option to set the peering type to `dynamic` or `static`.
  - If set to `static`, use the `--peers` option to list the URLs of all peers that the validator should connect to, using the format `tcp://`*hostname*`:`*port*. Specify multiple peer URLs in a comma-separated list.
  - If set to `dynamic`, any static peers will be processed first, before starting the topology buildout starting, then the URLs specified by `--seeds` will be used for the initial connection to the validator network.
- Use `--scheduler` to set the scheduler type to `serial` or `parallel`. Note that both scheduler types result in the same deterministic results and are completely interchangeable. However, parallel processing of transactions provides a performance improvement even for fast transaction workloads by reducing the overall latency effects that occur when transactions are processed serially.
- Use `--network-auth` to specify the required authorization procedure (`trust` or `challenge`) that validator connections must go through before they are allowed to participate on the network. To use network permissions, specify `challenge`, which requires connections to sign a challenge so their identity can be proved.

```
usage: sawtooth-validator [-h] [--config-dir CONFIG_DIR] [-B BIND]
                          [-P {static,dynamic}] [-E ENDPOINT] [-s SEEDS]
                          [-p PEERS] [-v] [--scheduler {serial,parallel}]
                          [--network-auth {trust,challenge}]
                          [--opentsdb-url OPENTSDB_URL]
                          [--opentsdb-db OPENTSDB_DB]
                          [--minimum-peer-connectivity MINIMUM_PEER_CONNECTIVITY]
                          [--maximum-peer-connectivity MAXIMUM_PEER_CONNECTIVITY]
                          [-V]

Configures and starts a Sawtooth validator.

optional arguments:
  -h, --help            show this help message and exit
  --config-dir CONFIG_DIR
                        specify the configuration directory
  -B BIND, --bind BIND  set the URL for the network or validator component service␣
↪endpoints with the format network:<endpoint> or component:<endpoint>. Use two --
↪bind options to specify both endpoints.
  -P {static,dynamic}, --peering {static,dynamic}
                        determine peering type for the validator: 'static' (must use -
↪-peers to list peers) or 'dynamic' (processes any static peers first, then starts␣
↪topology buildout).
  -E ENDPOINT, --endpoint ENDPOINT
                        specifies the advertised network endpoint URL
  -s SEEDS, --seeds SEEDS
                        provide URI(s) for the initial connection to the validator␣
↪network, in the format tcp://<hostname>:<port>. Specify multiple URIs in a comma-
↪separated list. Repeating the --seeds option is also accepted.
  -p PEERS, --peers PEERS
                        list static peers to attempt to connect to in the format tcp:/
↪/<hostname>:<port>. Specify multiple peers in a comma-separated list. Repeating the␣
↪--peers option is also accepted.
  -v, --verbose         enable more verbose output to stderr
```

(continues on next page)

(continued from previous page)

```
  --scheduler {serial,parallel}
                        set scheduler type: serial or parallel
  --network-auth {trust,challenge}
                        identify type of authorization required to join validator␣
↪network.
  --opentsdb-url OPENTSDB_URL
                        specify host and port for Open TSDB database              ␣
↪       used for metrics
  --opentsdb-db OPENTSDB_DB
                        specify name of database used for storing                 ␣
↪    metrics
  --minimum-peer-connectivity MINIMUM_PEER_CONNECTIVITY
                        set the minimum number of peers required before           ␣
↪          stopping peer search
  --maximum-peer-connectivity MAXIMUM_PEER_CONNECTIVITY
                        set the maximum number of peers to accept
  -V, --version         display version information
```

7.3 sawtooth-rest-api

The `sawtooth-rest-api` command starts the REST API and connects to the validator.

The REST API is designed to run alongside a validator, providing potential clients access to blockchain and state data through common HTTP/JSON standards. It is a stateless process, and does not store any part of the blockchain or blockchain state. Instead it acts as a go between, translating HTTP requests into validator requests, and sending back the results as JSON. As a result, running the Sawtooth REST API requires that a validator already be running and available over TCP.

Options for `sawtooth-rest-api` specify the bind address for the host and port (by default, `http://localhost:8008`) and the TCP address where the validator is running (the default is `tcp://localhost:4004`). An optional timeout value configures how long the REST API will wait for a response for the validator.

```
usage: sawtooth-rest-api [-h] [-B BIND] [-C CONNECT] [-t TIMEOUT] [-v]
                         [--opentsdb-url OPENTSDB_URL]
                         [--opentsdb-db OPENTSDB_DB] [-V]

Starts the REST API application and connects to a specified validator.

optional arguments:
  -h, --help            show this help message and exit
  -B BIND, --bind BIND  identify host and port for API to run on default:
                        http://localhost:8080)
  -C CONNECT, --connect CONNECT
                        specify URL to connect to a running validator
  -t TIMEOUT, --timeout TIMEOUT
                        set time (in seconds) to wait for validator response
  -v, --verbose         enable more verbose output to stderr
  --opentsdb-url OPENTSDB_URL
                        specify host and port for Open TSDB database used for
                        metrics
  --opentsdb-db OPENTSDB_DB
                        specify name of database for storing metrics
  -V, --version         display version information
```

7.4 sawadm

The `sawadm` command is used for Sawtooth administration tasks. The `sawadm` subcommands create validator keys during initial configuration and help create the genesis block when initializing a validator.

```
usage: sawadm [-h] [-v] [-V] {genesis,keygen} ...

Provides subcommands to create validator keys and create the genesis block

optional arguments:
  -h, --help        show this help message and exit
  -v, --verbose     enable more verbose output
  -V, --version     display version information

subcommands:
  {genesis,keygen}
    genesis         Creates the genesis.batch file for initializing the
                    validator
    keygen          Generates keys for the validator to use when signing
                    blocks
```

7.4.1 sawadm genesis

The `sawadm genesis` subcommand produces a file for use during the initialization of a validator. A network requires an initial block (known as the *genesis block*) whose signature will determine the blockchain ID. This initial block is produced from a list of batches, which will be applied at genesis time.

The optional argument *input_file* specifies one or more files containing serialized `BatchList` protobuf messages to add to the genesis data. (Use a space to separate multiple files.) If no input file is specified, `sawadm keygen` produces an empty genesis block.

The output is a file containing a serialized `GenesisData` protobuf message. This file, when placed at *sawtooth_data*/`genesis.batch`, will trigger the genesis process.

Note: The location of *sawtooth_data* depends on whether the environment variable `SAWTOOTH_HOME` is set. If it is, then *sawtooth_data* is located at `SAWTOOTH_HOME/data`. If it is not, then *sawtooth_data* is located at `/var/lib/sawtooth`.

When `sawadm genesis` runs, it displays the path and filename of the target file where the serialized `GenesisData` is written. (Default: *sawtooth_data*/`genesis.batch`.) For example:

```
$ sawadm genesis config.batch mktplace.batch
Generating /var/lib/sawtooth/genesis.batch
```

Use `--output` *filename* to specify a different name for the target file.

```
usage: sawadm genesis [-h] [-v] [-V] [-o OUTPUT] [input_file [input_file ...]]

Generates the genesis.batch file for initializing the validator.

positional arguments:
  input_file            file or files containing batches to add to the
                        resulting GenesisData

```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  -o OUTPUT, --output OUTPUT
                        choose the output file for GenesisData

This command generates a serialized GenesisData protobuf message and stores it
in the genesis.batch file. One or more input files (optional) can contain
serialized BatchList protobuf messages to add to the GenesisData. The output
shows the location of this file. By default, the genesis.batch file is stored
in /var/lib/sawtooth. If $SAWTOOTH_HOME is set, the location is
$SAWTOOTH_HOME/data/genesis.batch. Use the --output option to change the name
of the file.
```

7.4.2 sawadm keygen

The `sawadm keygen` subcommand generates keys that the validator uses to sign blocks. This system-wide key must be created during Sawtooth configuration.

Validator keys are stored in the directory `/etc/sawtooth/keys/`. By default, the public-private key files are named `validator.priv` and validator.pub. Use the <key-name> argument to specify a different file name.

```
usage: sawadm keygen [-h] [-v] [-V] [--force] [-q] [key_name]

Generates keys for the validator to use when signing blocks.

positional arguments:
  key_name       name of the key to create

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  enable more verbose output
  -V, --version  display version information
  --force        overwrite files if they exist
  -q, --quiet    do not display output

The private and public key pair is stored in /etc/sawtooth/keys/<key-
name>.priv and /etc/sawtooth/keys/<key-name>.pub.
```

7.5 sawnet

The `sawnet` command is used to interact with an entire network of Sawtooth nodes.

```
usage: sawnet [-h] [-v] [-V] {compare-chains,list-blocks,peers} ...

Inspect status of a sawtooth network

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
```

(continues on next page)

(continued from previous page)

```
subcommands:
  {compare-chains,list-blocks,peers}
    compare-chains      Compare chains from different nodes.
    list-blocks         List blocks from different nodes.
    peers               Shows the peering arrangment of a network
```

7.5.1 sawnet compare-chains

The `sawnet compare-chains` subcommand compares chains across the specified nodes.

```
usage: sawnet compare-chains [-h] [-v] [-V] [--users USERNAME[:PASSWORD]]
                             [-l LIMIT] [--table] [--tree]
                             urls [urls ...]

Compute and display information about how the chains at different nodes differ.

positional arguments:
  urls                  The URLs of the validator's REST APIs of interest,
                        separated by commas or spaces. (no default)

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --users USERNAME[:PASSWORD]
                        Specify the users to authorize requests, in the same
                        order as the URLs, separate by commas. Passing empty
                        strings between commas is supported.
  -l LIMIT, --limit LIMIT
                        the number of blocks to request at a time
  --table               Print out a fork table for all nodes since the common
                        ancestor.
  --tree                Print out a fork tree for all nodes since the common
                        ancestor.

By default, prints a table of summary data and a table of per-node data with
the following fields. Pass --tree for a fork graph.

COMMON ANCESTOR
    The most recent block that all chains have in common.

COMMON HEIGHT
    Let min_height := the minimum height of any chain across all nodes passed
    in. COMMON HEIGHT = min_height.

HEAD
    The block id of the most recent block on a given chain.

HEIGHT
    The block number of the most recent block on a given chain.

LAG
    Let max_height := the maximum height of any chain across all nodes passed
    in. LAG = max_height - HEIGHT for a given chain.
```

(continues on next page)

(continued from previous page)

```

DIVERG
    Let common_ancestor_height := the height of the COMMON ANCESTOR.
    DIVERG = HEIGHT - common_ancestor_height
```

7.5.2 sawnet peers

```
usage: sawnet peers [-h] {list,graph} ...

Shows the peering arrangment of a network.

optional arguments:
  -h, --help    show this help message and exit

subcommands:
  {list,graph}
    list        Lists peers for validators with given URLs
    graph       Generates a file to graph a network's peering arrangement
```

7.5.3 sawnet peers list

The `sawnet peers list` subcommand displays the peers of the specified nodes.

```
usage: sawnet peers list [-h] [-v] [-V] [--users USERNAME[:PASSWORD]]
                         [--pretty]
                         urls [urls ...]

Lists peers for validators with given URLs.

positional arguments:
  urls                  The URLs of the validator's REST APIs of interest,
                        separated by commas or spaces. (no default)

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --users USERNAME[:PASSWORD]
                        Specify the users to authorize requests, in the same
                        order as the URLs, separate by commas. Passing empty
                        strings between commas is supported.
  --pretty, -p          Pretty-print the results
```

7.5.4 sawnet peers graph

The `sawnet peers graph` subcommand displays a file called `peers.dot` that describes the peering arrangement of the specified nodes.

```
usage: sawnet peers graph [-h] [-v] [-V] [--users USERNAME[:PASSWORD]]
                          [-o OUTPUT] [--force]
                          urls [urls ...]
```

(continues on next page)

(continued from previous page)

```

Generates a file to graph a network's peering arrangement.

positional arguments:
  urls                  The URLs of the validator's REST APIs of interest,
                        separated by commas or spaces. (no default)

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --users USERNAME[:PASSWORD]
                        Specify the users to authorize requests, in the same
                        order as the URLs, separate by commas. Passing empty
                        strings between commas is supported.
  -o OUTPUT, --output OUTPUT
                        The path of the dot file to be produced (defaults to
                        peers.dot)
  --force               TODO
```

7.6 sawset

The `sawset` command is used to work with settings proposals.

Sawtooth supports storing settings on-chain. The `sawset` subcommands can be used to view the current proposals, create proposals, vote on existing proposals, and produce setting values that will be set in the genesis block.

```
usage: sawset [-h] [-v] [-V] {genesis,proposal} ...

Provides subcommands to change genesis block settings and to view, create, and
vote on settings proposals.

optional arguments:
  -h, --help          show this help message and exit
  -v, --verbose       enable more verbose output
  -V, --version       display version information

subcommands:
  {genesis,proposal}
    genesis           Creates a genesis batch file of settings transactions
    proposal          Views, creates, or votes on settings change proposals
```

7.6.1 sawset genesis

The `sawset genesis` subcommand creates a Batch of settings proposals that can be consumed by `sawadm genesis` and used during genesis block construction.

```
usage: sawset genesis [-h] [-k KEY] [-o OUTPUT] [-T APPROVAL_THRESHOLD]
                      [-A AUTHORIZED_KEY]

Creates a Batch of settings proposals that can be consumed by "sawadm genesis"
and used during genesis block construction.
```

(continues on next page)

(continued from previous page)

```

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     specify signing key for resulting batches and initial
                        authorized key
  -o OUTPUT, --output OUTPUT
                        specify the output file for the resulting batches
  -T APPROVAL_THRESHOLD, --approval-threshold APPROVAL_THRESHOLD
                        set the number of votes required to enable a setting
                        change
  -A AUTHORIZED_KEY, --authorized-key AUTHORIZED_KEY
                        specify a public key for the user authorized to submit
                        config transactions
```

7.6.2 sawset proposal

The Settings transaction family supports a simple voting mechanism for applying changes to on-change settings. The `sawset proposal` subcommands provide tools to view, create and vote on proposed settings.

```
usage: sawset proposal [-h] {create,list,vote} ...

Provides subcommands to view, create, or vote on proposed settings

optional arguments:
  -h, --help          show this help message and exit

subcommands:
  {create,list,vote}
    create            Creates proposals for setting changes
    list              Lists the currently proposed (not active) settings
    vote              Votes for specific setting change proposals
```

7.6.3 sawset proposal create

The `sawset proposal create` subcommand creates proposals for settings changes. The change may be applied immediately or after a series of votes, depending on the vote threshold setting.

```
usage: sawset proposal create [-h] [-k KEY] [-o OUTPUT | --url URL]
                              setting [setting ...]

Create proposals for settings changes. The change may be applied immediately
or after a series of votes, depending on the vote threshold setting.

positional arguments:
  setting               configuration setting as key/value pair with the
                        format <key>=<value>

optional arguments:
  -h, --help            show this help message and exit
  -k KEY, --key KEY     specify a signing key for the resulting batches
  -o OUTPUT, --output OUTPUT
                        specify the output file for the resulting batches
  --url URL             identify the URL of a validator's REST API
```

7.6.4 sawset proposal list

The `sawset proposal list` subcommand displays the currently proposed settings that are not yet active. This list of proposals can be used to find proposals to vote on.

```
usage: sawset proposal list [-h] [--url URL] [--public-key PUBLIC_KEY]
                            [--filter FILTER]
                            [--format {default,csv,json,yaml}]

Lists the currently proposed (not active) settings. Use this list of proposals
to find proposals to vote on.

optional arguments:
  -h, --help            show this help message and exit
  --url URL             identify the URL of a validator's REST API
  --public-key PUBLIC_KEY
                        filter proposals from a particular public key
  --filter FILTER       filter keys that begin with this value
  --format {default,csv,json,yaml}
                        choose the output format
```

7.6.5 sawset proposal vote

The `sawset proposal vote` subcommand votes for a specific settings-change proposal. Use `sawset proposal list` to find the proposal id.

```
usage: sawset proposal vote [-h] [--url URL] [-k KEY]
                            proposal_id {accept,reject}

Votes for a specific settings change proposal. Use "sawset proposal list" to
find the proposal id.

positional arguments:
  proposal_id        identify the proposal to vote on
  {accept,reject}    specify the value of the vote

optional arguments:
  -h, --help         show this help message and exit
  --url URL          identify the URL of a validator's REST API
  -k KEY, --key KEY  specify a signing key for the resulting transaction batch
```

7.7 poet

The `poet` command initializes the Proof of Elapsed Time (PoET) consensus mechanism for Sawtooth by generating enclave setup information and creating a Batch for the genesis block. For more information, see *PoET 1.0 Specification*.

The `poet` command provides subcommands for configuring a node to use Sawtooth with the PoET consensus method.

```
usage: poet [-h] [-v] [-V] {registration,enclave} ...

Provides subcommands for creating PoET registration.

```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information

subcommands:
  {registration,enclave}
    registration        Provides a subcommand for creating PoET registration
    enclave             Generates enclave setup information
```

7.7.1 poet registration

The `poet registration` subcommand provides a command to work with the PoET validator registry.

```
usage: poet registration [-h] {create} ...

Provides a subcommand for creating PoET registration.

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  {create}
    create    Creates a batch to enroll a validator in the validator registry
```

7.7.2 poet registration create

The `poet registration create` subcommand creates a batch to enroll a validator in the network's validator registry. It must be run from the validator host wishing to enroll.

```
usage: poet registration create [-h] [--enclave-module {simulator,sgx}]
                                [-k KEY] [-o OUTPUT] [-b BLOCK]

Creates a batch to enroll a validator in the validator registry.

optional arguments:
  -h, --help            show this help message and exit
  --enclave-module {simulator,sgx}
                        configure the enclave module to use
  -k KEY, --key KEY     identify file containing transaction signing key
  -o OUTPUT, --output OUTPUT
                        change default output file name for resulting batches
  -b BLOCK, --block BLOCK
                        specify the most recent block identifier to use as a
                        sign-up nonce
```

7.7.3 poet enclave

The `poet enclave` subcommand generates enclave setup information.

```
usage: poet enclave [-h] [--enclave-module {simulator,sgx}]
                    {measurement,basename}

Generates enclave setup information.

positional arguments:
  {measurement,basename}
                        enclave characteristic to retrieve

optional arguments:
  -h, --help            show this help message and exit
  --enclave-module {simulator,sgx}
                        identify the enclave module to query
```

7.8 identity-tp

The `identity-tp` command starts the Identity transaction processor, which handles on-chain permissioning for transactor and validator keys to streamline managing identities for lists of public keys.

The Settings transaction processor is required when using the Identity transaction processor.

In order to send identity transactions, your public key must be stored in `sawtooth.identity.allowed_keys`.

```
usage: identity-tp [-h] [-C CONNECT] [-v] [-V]

Starts an Identity transaction processor.

optional arguments:
  -h, --help            show this help message and exit
  -C CONNECT, --connect CONNECT
                        specify the endpoint for the validator connection
  -v, --verbose         enable more verbose output to stderr
  -V, --version         display version information

This process is required to apply any changes to on-chain permissions used by the␣
↪Sawtooth platform.
```

7.9 intkey

The `intkey` command starts the IntegerKey transaction processor, which provides functions that can be used to test deployed ledgers.

The `intkey` command provides subcommands to set, increment, and decrement the value of entries stored in a state dictionary.

```
usage: intkey [-h] [-v] [-V]
              {set,inc,dec,show,list,generate,load,populate,create_batch,workload}
              ...

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
```

(continues on next page)

(continued from previous page)

```
subcommands:
  {set,inc,dec,show,list,generate,load,populate,create_batch,workload}
    set                 Sets an intkey value
    inc                 Increments an intkey value
    dec                 Decrements an intkey value
    show                Displays the specified intkey value
    list                Displays all intkey values
```

7.9.1 intkey set

The `intkey set` subcommand sets a key (*name*) to the specified value. This transaction will fail if the value is less than 0 or greater than 2^{32} - 1.

```
usage: intkey set [-h] [-v] [-V] [--url URL] [--keyfile KEYFILE]
                  [--wait [WAIT]]
                  name value

Sends an intkey transaction to set <name> to <value>.

positional arguments:
  name               name of key to set
  value              amount to set

optional arguments:
  -h, --help         show this help message and exit
  -v, --verbose      enable more verbose output
  -V, --version      display version information
  --url URL          specify URL of REST API
  --keyfile KEYFILE  identify file containing user's private key
  --wait [WAIT]      set time, in seconds, to wait for transaction to commit
```

7.9.2 intkey inc

The `intkey inc` subcommand increments a key (*name*) by the specified value. This transaction will fail if the key is not set or if the resulting value would exceed 2^{32} - 1.

```
usage: intkey inc [-h] [-v] [-V] [--url URL] [--keyfile KEYFILE]
                  [--wait [WAIT]]
                  name value

Sends an intkey transaction to increment <name> by <value>.

positional arguments:
  name               identify name of key to increment
  value              specify amount to increment

optional arguments:
  -h, --help         show this help message and exit
  -v, --verbose      enable more verbose output
  -V, --version      display version information
  --url URL          specify URL of REST API
```

(continues on next page)

(continued from previous page)

```
  --keyfile KEYFILE  identify file containing user's private key
  --wait [WAIT]      set time, in seconds, to wait for transaction to commit
```

7.9.3 intkey dec

The `intkey dec` subcommand decrements a key (*name*) by the specified value. This transaction will fail if the key is not set or if the resulting value would be less than 0.

```
usage: intkey dec [-h] [-v] [-V] [--url URL] [--keyfile KEYFILE]
                  [--wait [WAIT]]
                  name value

Sends an intkey transaction to decrement <name> by <value>.

positional arguments:
  name               identify name of key to decrement
  value              amount to decrement

optional arguments:
  -h, --help         show this help message and exit
  -v, --verbose      enable more verbose output
  -V, --version      display version information
  --url URL          specify URL of REST API
  --keyfile KEYFILE  identify file containing user's private key
  --wait [WAIT]      set time, in seconds, to wait for transaction to commit
```

7.9.4 intkey show

The `intkey show` subcommand displays the value of the specified key (*name*).

```
usage: intkey show [-h] [-v] [-V] [--url URL] name

Shows the value of the key <name>.

positional arguments:
  name           name of key to show

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  enable more verbose output
  -V, --version  display version information
  --url URL      specify URL of REST API
```

7.9.5 intkey list

The `intkey list` subcommand displays the value of all keys.

```
usage: intkey list [-h] [-v] [-V] [--url URL]

Shows the values of all keys in intkey state.

```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  enable more verbose output
  -V, --version  display version information
  --url URL      specify URL of REST API
```

7.10 settings-tp

The `settings-tp` command starts the Settings transaction processor, which provides a methodology for storing on-chain configuration settings.

Note: This transaction processor is required in order to apply changes to on-chain settings.

The settings stored in state as a result of this transaction family play a critical role in the operation of a validator. For example, the consensus module uses these settings. In the case of PoET, one cross-network setting is target wait time (which must be the same across validators), that is stored as `sawtooth.poet.target_wait_time`. Other parts of the system use these settings similarly; for example, the list of enabled transaction families is used by the transaction processing platform. In addition, pluggable components such as transaction family implementations can use the settings during their execution.

This design supports two authorization options: either a single authorized key that can make changes or multiple authorized keys. In the case of multiple keys, a percentage of votes signed by the keys is required to make a change. Note that only the keys in `sawtooth.settings.vote.authorized_keys` are allowed to submit setting transactions.

```
usage: settings-tp [-h] [-C CONNECT] [-v] [-V]

Starts a Settings transaction processor (settings-tp).

optional arguments:
  -h, --help            show this help message and exit
  -C CONNECT, --connect CONNECT
                        specify the endpoint for the validator connection
                        (default: tcp://localhost:4004)
  -v, --verbose         enable more verbose output to stderr
  -V, --version         display version information

This process is required to apply any changes to on-chain settings used by the␣
↪Sawtooth platform.
```

7.11 xo

The `xo` command starts the XO transaction processor.

This command demonstrates an example client that uses the XO transaction family to play a simple game of Tic-tac-toe (also known as Noughts and Crosses, or X's and O's). This command handles the construction and submission of transactions to a running validator via the URL of the validator's REST API.

Before playing a game, you must start a validator, the XO transaction processor,and the REST API. The XO client sends requests to update and query the blockchain to the URL of the REST API (by default, `http://127.0.0.1:8080`).

For more information on requirements and game rules, see *Introduction to the XO Transaction Family*.

The `xo` command provides subcommands for playing XO on the command line.

```
usage: xo [-h] [-v] [-V] {create,list,show,take,delete} ...

Provides subcommands to play tic-tac-toe (also known as Noughts and Crosses)
by sending XO transactions.

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information

subcommands:
  {create,list,show,take,delete}
    create              Creates a new xo game
    list                Displays information for all xo games
    show                Displays information about an xo game
    take                Takes a space in an xo game
```

7.11.1 xo create

The `xo create` subcommand starts an XO game with the specified name.

```
usage: xo create [-h] [-v] [-V] [--url URL] [--username USERNAME]
                 [--key-dir KEY_DIR] [--auth-user AUTH_USER]
                 [--auth-password AUTH_PASSWORD] [--disable-client-validation]
                 [--wait [WAIT]]
                 name

Sends a transaction to start an xo game with the identifier <name>. This
transaction will fail if the specified game already exists.

positional arguments:
  name                  unique identifier for the new game

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --url URL             specify URL of REST API
  --username USERNAME   identify name of user's private key file
  --key-dir KEY_DIR     identify directory of user's private key file
  --auth-user AUTH_USER
                        specify username for authentication if REST API is
                        using Basic Auth
  --auth-password AUTH_PASSWORD
                        specify password for authentication if REST API is
                        using Basic Auth
  --disable-client-validation
                        disable client validation
  --wait [WAIT]         set time, in seconds, to wait for game to commit
```

7.11.2 xo list

The `xo list` subcommand displays information for all XO games in state.

```
usage: xo list [-h] [-v] [-V] [--url URL] [--username USERNAME]
               [--key-dir KEY_DIR] [--auth-user AUTH_USER]
               [--auth-password AUTH_PASSWORD]

Displays information for all xo games in state, showing the players, the game
state, and the board for each game.

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --url URL             specify URL of REST API
  --username USERNAME   identify name of user's private key file
  --key-dir KEY_DIR     identify directory of user's private key file
  --auth-user AUTH_USER
                        specify username for authentication if REST API is
                        using Basic Auth
  --auth-password AUTH_PASSWORD
                        specify password for authentication if REST API is
                        using Basic Auth
```

7.11.3 xo show

The `xo show` subcommand displays information about the specified XO game.

```
usage: xo show [-h] [-v] [-V] [--url URL] [--username USERNAME]
               [--key-dir KEY_DIR] [--auth-user AUTH_USER]
               [--auth-password AUTH_PASSWORD]
               name

Displays the xo game <name>, showing the players, the game state, and the
board

positional arguments:
  name                  identifier for the game

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --url URL             specify URL of REST API
  --username USERNAME   identify name of user's private key file
  --key-dir KEY_DIR     identify directory of user's private key file
  --auth-user AUTH_USER
                        specify username for authentication if REST API is
                        using Basic Auth
  --auth-password AUTH_PASSWORD
                        specify password for authentication if REST API is
                        using Basic Auth
```

7.11.4 xo take

The `xo take` subcommand makes a move in an XO game by sending a transaction to take the identified space. This transaction will fail if the game *name* does not exist, if it is not the sender's turn, or if *space* is already taken.

```
usage: xo take [-h] [-v] [-V] [--url URL] [--username USERNAME]
               [--key-dir KEY_DIR] [--auth-user AUTH_USER]
               [--auth-password AUTH_PASSWORD] [--wait [WAIT]]
               name space

Sends a transaction to take a square in the xo game with the identifier
<name>. This transaction will fail if the specified game does not exist.

positional arguments:
  name                  identifier for the game
  space                 number of the square to take (1-9); the upper-left
                        space is 1, and the lower-right space is 9

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --url URL             specify URL of REST API
  --username USERNAME   identify name of user's private key file
  --key-dir KEY_DIR     identify directory of user's private key file
  --auth-user AUTH_USER
                        specify username for authentication if REST API is
                        using Basic Auth
  --auth-password AUTH_PASSWORD
                        specify password for authentication if REST API is
                        using Basic Auth
  --wait [WAIT]         set time, in seconds, to wait for take transaction to
                        commit
```

7.11.5 xo delete

The `xo delete` subcommand deletes an existing xo game.

```
usage: xo delete [-h] [-v] [-V] [--url URL] [--username USERNAME]
                 [--key-dir KEY_DIR] [--auth-user AUTH_USER]
                 [--auth-password AUTH_PASSWORD] [--wait [WAIT]]
                 name

positional arguments:
  name                  name of the game to be deleted

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         enable more verbose output
  -V, --version         display version information
  --url URL             specify URL of REST API
  --username USERNAME   identify name of user's private key file
  --key-dir KEY_DIR     identify directory of user's private key file
  --auth-user AUTH_USER
                        specify username for authentication if REST API is
                        using Basic Auth
  --auth-password AUTH_PASSWORD
                        specify password for authentication if REST API is
                        using Basic Auth
  --wait [WAIT]         set time, in seconds, to wait for delete transaction
                        to commit
```

CHAPTER

EIGHT

COMMUNITY

Welcome to the Sawtooth community!

For help topics, we recommend joining us on Chat (link below).

8.1 Joining the Discussion

8.1.1 Chat

Hyperledger Sawtooth RocketChat is the place to go for real-time chat about everything from quick help to involved discussions.

For general Hyperledger Sawtooth discussions:

> #sawtooth

For Hyperledger Sawtooth Consensus discussions:

> #sawtooth-consensus

8.1.2 Mailing Lists

The Hyperledger Sawtooth mailing list is hosted by the Hyperledger Project:

> hyperledger-stl Mailing List

8.2 Tracking Issues

A great way to contribute is by reporting issues. Before reporting an issue, please review the current open issues to see if someone has already reported the issue.

8.2.1 Using JIRA

Hyperledger Sawtooth uses JIRA as our issue tracking system:

https://jira.hyperledger.org/projects/STL

If you want to contribute to Hyperledger Sawtooth quickly, we have a list of issues that will help you get involved right away. See the open Sawtooth issues:

https://jira.hyperledger.org/issues/?filter=10612

8.2.2 How to Report an Issue

To report issues, log into jira.hyperledger.org, which requires a Linux Foundation Account.

Create issues in JIRA under the Hyperledger Sawtooth project, which uses the `STL` JIRA key.

When reporting an issue, please provide as much detail as possible about how to reproduce it. If possible, explain how to reproduce the issue. Details are very helpful. Please include the following information:

- OS version
- Sawtooth version
- Environment details (virtual, physical, etc.)
- Steps to reproduce the issue
- Actual results
- Expected results

If you would like, you could also describe the issue on RocketChat (see *Joining the Discussion*) for initial feedback before submitting the issue in JIRA.

8.3 Contributing

8.3.1 Ways to Contribute to Hyperledger Sawtooth

Contributions from the development community help improve the capabilities of Hyperledger Sawtooth. These contributions are the most effective way to make a positive impact on the project.

Ways you can contribute:

- Bugs or issues: Report problems or defects found when working with Sawtooth
- Core features and enhancements: Provide expanded capabilities or optimizations
- Documentation: Improve existing documentation or create new information
- Tests for events and results: Add functional, performance, or scalability tests

Hyperledger Sawtooth issues can be found in *Using JIRA*. Any unassigned items are probably still open. When in doubt, ask on RocketChat about a specific JIRA issue (see *Joining the Discussion*).

8.3.2 The Commit Process

Hyperledger Sawtooth is Apache 2.0 licensed and accepts contributions via GitHub pull requests. When contributing code, please follow these guidelines:

- Fork the repository and make your changes in a feature branch
- Include unit and integration tests for any new features and updates to existing tests
- Ensure that the unit and integration tests run successfully. Run both of these tests with `./bin/run_tests`
- Check that the lint tests pass by running `./bin/run_lint -s master`.

Pull Request Guidelines

A pull request can contain a single commit or multiple commits. The most important guideline is that a single commit should map to a single fix or enhancement. Here are some example scenarios:

- If a pull request adds a feature but also fixes two bugs, the pull request should have three commits: one commit for the feature change and two commits for the bug fixes.
- If a PR is opened with five commits that contain changes to fix a single issue, the PR should be rebased to a single commit.
- If a PR is opened with several commits, where the first commit fixes one issue and the rest fix a separate issue, the PR should be rebased to two commits (one for each issue).

Important: Your pull request should be rebased against the current master branch. Do not merge the current master branch in with your topic branch. Do not use the Update Branch button provided by GitHub on the pull request page.

Commit Messages

Commit messages should follow common Git conventions, such as using the imperative mood, separate subject lines, and a line length of 72 characters. These rules are well documented in Chris Beam's blog post.

Signed-off-by

Each commit must include a "Signed-off-by" line in the commit message (`git commit -s`). This sign-off indicates that you agree the commit satisfies the Developer Certificate of Origin (DCO).

Commit Email Address

Your commit email address must match your GitHub email address. For more information, see https://help.github.com/articles/setting-your-commit-email-address-in-git/

Important GitHub Requirements

A pull request cannot merged until it has passed these status checks:

- The build must pass on Jenkins
- The PR must be approved by at least two reviewers without any outstanding requests for changes

Integrating GitHub Commits with JIRA

You can link JIRA issues to your commits, which will integrate developer activity with the associated issue. JIRA uses the issue key to associate the commit with the issue, so that the commit can be summarized in the development panel for the JIRA issue.

When you make a commit, add the JIRA issue key to the end of the commit message or to the branch name. Either method should integrate your commit with the JIRA issue that it references.

8.4 Code of Conduct

When participating, please be respectful and courteous.

Hyperledger Sawtooth uses the Hyperledger Project Code of Conduct.

CHAPTER

NINE

GLOSSARY

This glossary defines general Sawtooth terms and concepts.

Batch Group of related transactions. In Sawtooth, a batch is the atomic unit of state change for the blockchain. A batch can contain one or more transactions. For a batch with multiple transactions, if one transaction fails, all transactions in that batch fail. (The client application is responsible for handling failure appropriately.) For more information, see *Transactions and Batches*.

Blockchain Distributed ledger that records transactions, in chronological order, shared by all participants in a Sawtooth network. Each block on the blockchain is linked by a cryptographic hash to the previous block.

Consensus Process of building agreement among a group of mutually distrusting participants (other nodes on a Sawtooth network). Sawtooth allows different types of consensus on the same blockchain. See also *Dynamic consensus*.

Core See *Sawtooth core*.

Distributed ledger See *Blockchain*.

Dynamic consensus Ability to change the blockchain consensus protocol for a running Sawtooth network. The current consensus is an on-chain setting, so it can be changed by submitting a transaction. For more information, see *Dynamic Consensus Algorithms*.

Genesis block First block on the blockchain. The genesis block initializes the Sawtooth network.

Global state Database that stores a local (validator-specific) record of transactions for the blockchain. Sawtooth represents state in a single instance of a Merkle-Radix tree on each validator node. For more information, see *Global State*.

Identity Sample transaction family that handles on-chain permissions (settings stored on the blockchain) for transactor and validator keys. This transaction family demonstrates how to streamline managing identities for lists of public keys. For more information, see *Identity Transaction Family*.

IntegerKey Sample transaction family with only three operations (set, increment, and decrement) that can be used to test deployed ledgers. For more information, see *IntegerKey Transaction Family*.

Journal Sawtooth core component that is responsible for maintaining and extending the blockchain for the validator. For more information, see *Journal*.

Merkle-Radix tree Addressable data structure that stores state data. A Merkle-Radix tree combines the benefits of a Merkle tree (also called a “hash tree”), which stores successive node hashes from leaf-to-root upon any changes to the tree, and a Radix tree, which has addresses that uniquely identify the paths to leaf nodes where information is stored. For more information, see *Merkle-Radix Tree Overview*.

Node Participant in Sawtooth network. Each node runs a single validator, a REST API, and one or more transaction processors.

Off-chain setting Setting or value that is stored locally, rather than on the blockchain.

On-chain setting Setting or value that is stored on the blockchain (also referred to as "in state") so that all participants on the network can access that information.

Permissioned network Restricted network of Sawtooth nodes. A permissioned network typically includes multiple parties with a mutual interest but without the mutual trust found in a network controlled by a single company or entity.

The blockchain stores the settings that specify permissions, such as roles and identities, so that all participants in the network can access this information.

PoET Proof of Elapsed Time, a Nakamoto-style consensus algorithm that is designed to support large networks. PoET relies on a Trusted Execution Environment (TEE) such as Intel® Software Guard Extensions (SGX). For more information, see *PoET 1.0 Specification*.

REST API In Sawtooth, a core component that adapts communication with a validator to HTTP/JSON standards. Sawtooth includes a REST API that is used by clients such as the Sawtooth CLI commands. Developers can use this REST API or develop custom APIs for client-validator communication. For more information, see *REST API*.

Sawtooth core Central Sawtooth software that is responsible for message handling, block validation and publishing, consensus, and global state management. The Sawtooth architecture separates these core functions from application-specific business logic, which is is handled by transaction families.

Sawtooth network Peer-to-peer network of nodes running a validator (and associated components) that are working on the same blockchain.

Settings Sample transaction family that provides a reference implementation for storing on-chain configuration settings. For more information, see *Settings Transaction Family*.

State See *Global state*.

State delta Result of a single change for a specific address in global state.

State delta subscriber Client framework that subscribes to a validator for state deltas (changes) for a specific set of transaction families. Usually, an application subscribes to state deltas for the purpose of off-chain storage or action, such as handling the failure of a transaction appropriately.

Transaction Function that changes the state of the blockchain. Each transaction is put into a Batch, either alone or with other related transactions, then sent to the validator for processing. For more information, see *Transactions and Batches*.

Transaction family Application-specific business logic that defines a set of operations or transaction types that are allowed on the blockchain. Sawtooth transaction families separate the transaction rules and content from the Sawtooth core functionality.

A transaction family implements a data model and transaction language for an application. Sawtooth includes example transaction families in several languages, such as Python, Go, and Java. For more information, see *Sample Transaction Families*.

Transaction processor Validates transactions and updates state based on the rules defined by the associated transaction family. Sawtooth includes transaction processors for the sample transaction families, such as `identity-tp` for the Identity transaction family. For more information, see *Transaction Family Specifications*.

Validator Component responsible for validating batches of transactions, combining them into blocks, maintaining consensus with the Sawtooth network, and coordinating communication between clients, transaction processors, and other validator nodes.

XO Sample transaction family that demonstrates basic transactions by playing tic-tac-toe on the blockchain. For more information, see *XO Transaction Family*.

HTTP ROUTING TABLE

PYTHON MODULE INDEX

INDEX