Ledger Service Design Document

Date: 8/30/2021
Author: Eric Gieseke

Introduction

Blockchain technology allows the creation of distributed trust networks that promise to disrupt many existing industries where trust has traditionally been provided through intermediaries (e.g., Banks, Insurance Companies).

This design specifies the implementation of a blockchain Ledger Service. The Ledger Service manages the transactions, accounts, and blocks that make up the Blockchain. Users submit transactions which once validated are added to a block. As Blocks fill up with Transactions, Account balances are updated, and the Blocks are added to the Ledger. Once committed to the Ledger, a Block, and the contained Transactions and Account balances are immutable. To ensure the immutability of the blocks, the blocks are chained together by including the hash of the previous block as a field in each new block. The blockchain can be validated at any time by recomputing the hashes of each block and comparing the result with the hash that has been stored in the next block.

Blockchain Properties

- Blockchain is a public ledger
- A sequence of transactions organized in blocks, guaranteeing three fundamental properties:
    - Everybody can read every block, so the blocks become common knowledge
    - Everybody can write a transaction in a future block
    - No one can alter the transactions in a block or the order of the blocks

Requirements

This section defines the requirements for the Ledger System.

Transaction Processing

1. The Ledger Service should accept transactions for inclusion into the blockchain.
2. Transactions should only be accepted if the paying and receiving addresses are linked to valid accounts.
3. Transactions should only be accepted if the paying account has a sufficient balance to cover the amount and associated transaction fee.
4. Once accepted, a transaction will be assigned a unique transaction id and added to a Block.

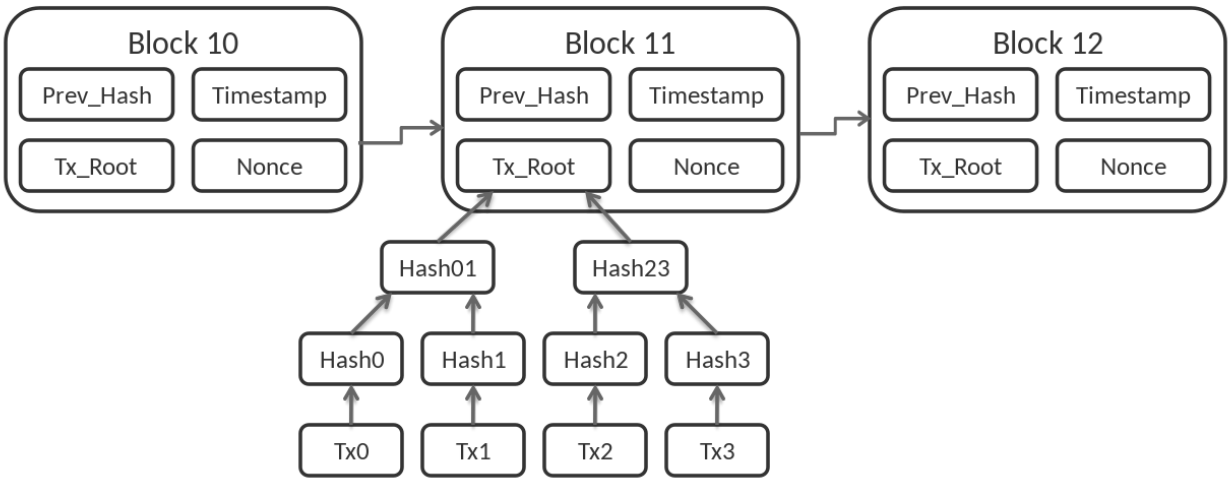
5. All transactions that are accepted require a fee of at least 10 units taken from the payer's account. The fees are transferred to the master account.

Accounts

1. The Ledger will manage a list of accounts.
2. When the blockchain is initialized, a master account is created that contains all currency, 2147483647 units.
3. Each new account is assigned a unique address.
4. All accounts have a balance that can be credited or debited through transactions.
5. The minimum balance for accounts is 0 and the maximum account balance is 2147483647 (i.e., `Integer.MAX_VALUE`).
6. The initial account balance for a new account is 0.

Blocks

1. Blocks are collections of transactions.
2. Each block contains exactly 10 transactions.
3. Each block contains a header that includes the hash of the previous block using the Sha-256 algorithm to compute the hash based on a Merkle tree of the contained transactions. (https://en.wikipedia.org/wiki/Merkle_tree)
4. Each block contains a map of all accounts and their balances which reflect any transactions contained within the block.



Example sub sequence of blocks showing the Merkle tree for block 11.

Ledger Service

The Ledger Service will support the following set of commands:

1. Initialize ledger
2. Accept Transaction
3. Get Account Balance for a single Account
4. Get Account Balance for all Accounts
5. Create Account
6. Get Transaction for a transaction Id
7. Get All Transactions For a Block
8. Get the total number of blocks.
9. Get Block by block number.

Not required

Persistence

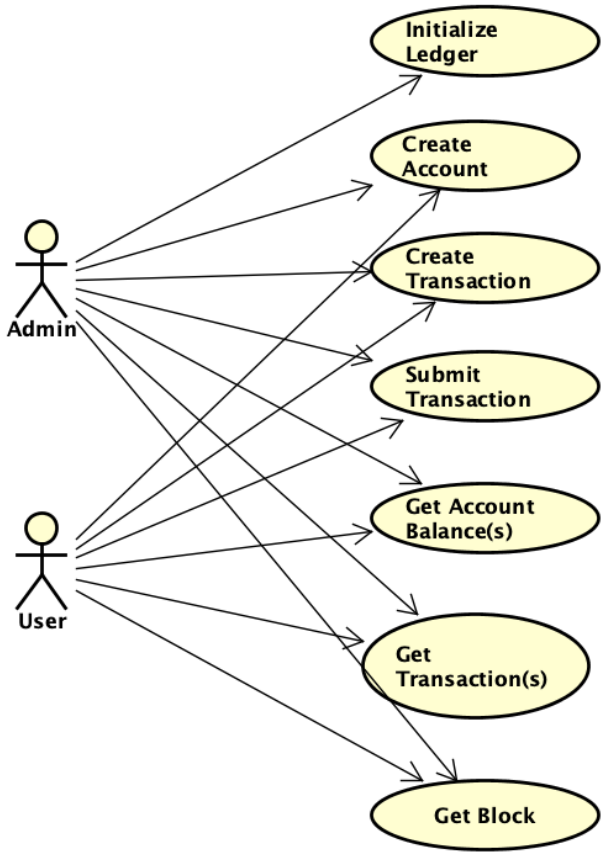
To limit the scope of this initial implementation, the Ledger will be maintained in memory. Persistence will not be required beyond keeping the Ledger in memory.

Transaction Signing

Normally, transactions would need to be signed by the Payer to prevent someone other than the owner of the account from creating transactions on their behalf. Again, to reduce the scope of the implementation, transaction signing is not required.

Use Cases

The following Use Case diagram describes the use cases supported by the Ledger System.



Ledger Service UML Use Case Diagram

Actors:

The actors of the Ledger System include Users, the Admin, and the Ledger Service.

Admin

The Admin is responsible for managing the Ledger, he is able to create and initialize the ledger, The Admin also performs the use cases available to Users.

Ledger Service

The autonomous Ledger Service performs the processing of transactions, maintaining account balances, and managing the integrity of the blockchain.

Use Cases:

Initialize Ledger

The Ledger's initialization includes setting the name, description, and seed for the blockchain. In addition to creating the genesis block, and the master account and initializing the master account balance.

Create Transaction

Creating a transaction will initialize a transaction object with the amount, fee, payer and receiver accounts.

Submit Transaction

Submitting a transaction will submit a created transaction to the blockchain. The amount of the transaction will be transferred from the Payer's account to the Receiver's account. The transaction fee will be transferred from the Payer's account to the master account. If the Payer's account does not have sufficient funds to support the transfers, or either the Payer or receiver account address is invalid, then the transaction will be rejected.

Create Account

Creating an account will initialize a new account with the provided unique account name and set the balance of the account to 0. The exception is when the master account is created during the Ledger Service initialization, in which case the account is funded with all available funds. If the provided account name is already used by an existing account, then an exception is returned.

Get Account Balance(s)

Return the balance for a given account, or a map of account names and balances if no account is specified.

Get Transactions(s)

Return the Transaction for the given transaction id.

Get Block

Return the Block for the given block number. Include the list of transactions, and also a map of account balances stored with the Block.

Process Transaction

Accept and process a transaction. First validate the transaction, checking that the payer and receiver accounts exist, the payer account has sufficient funds to cover the amount and the fee, and that the fee is at least 10 units. Once validated, add the transaction to the current block and adjust the payer, receiver account balances to reflect the transfer of the amount. Also, update the payer and master account to reflect the transfer of the fee from the payer to the master account. If after adding the transaction, the block is full, then compute the hash for the block,

and store the hash in the hash field of the block, and create a new block ready to receive the next block.

Manage Ledger

Make sure that the Ledger is consistent and transactions are processed one at a time in the order that they are received. Ensure that transactions are processed only once, the same transaction should not appear in the ledger more than once. Ensure that the hashes that link the blocks are consistent.

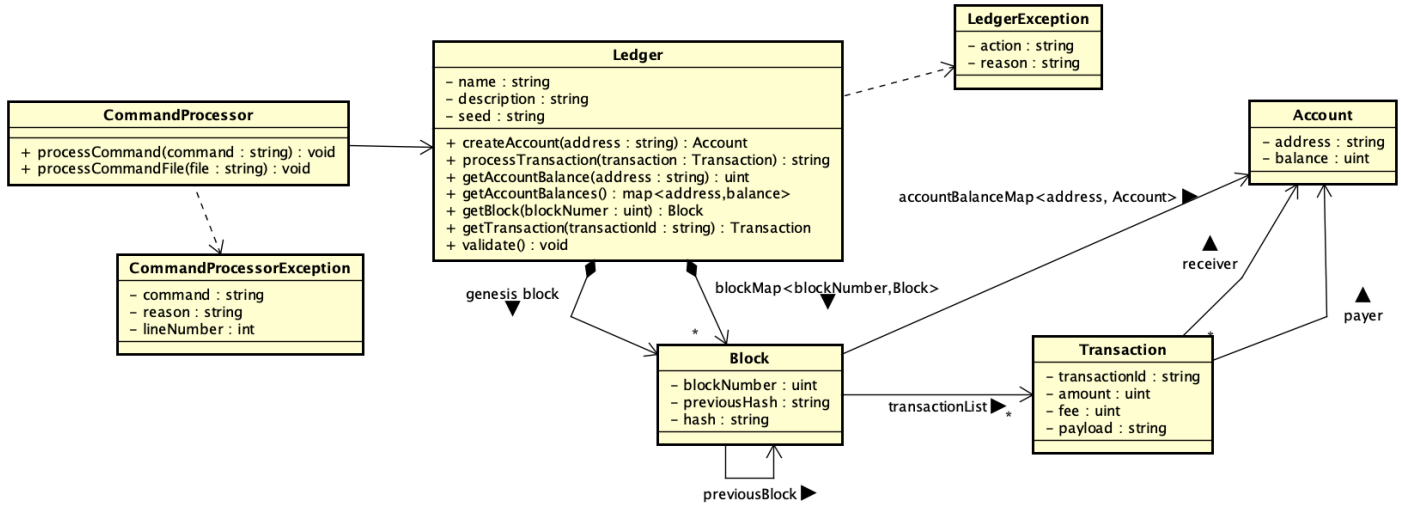
Maintain Account Balances

As new accounts are created and transactions are processed, make sure that the account balances are maintained accurately. After a block is saved, the total value of all accounts in the account balances map should equal the maximum possible account value of 2147483647 (i.e., `Integer.MAX_VALUE`).

Implementation

Class Diagram

The following class diagram defines the Ledger implementation classes contained within the package “com.cscie97.ledger”.



Ledger Service UML Class Diagram

Class Dictionary

This section specifies the class dictionary for the Ledger System. The classes should be defined within the package “com.cscie97.ledger”.

Account

The Account class represents an individual account within the Ledger Service. An account contains an address that provides a unique identity for the Account. The Account also contains a balance that represents the value of the account. The account can only be updated by the Ledger Service.

Properties

Property Name	Type	Description
address	string	Unique identifier for the account, assigned automatically when an Account instance is created.

balance	Unsigned integer	Balance of the account which reflects the total transfers to and from the Account, including fees for transactions where the account is the payer.

Transaction

The Transaction class represents a transaction in the Ledger System. A transaction contains a transaction id, an amount, a fee, a note, and references a payer account and a receiver account. The transaction amount is transferred from the payer's account balance to the receiver's account balance. The transaction fee is transferred from the payer's account to the master account. Transactions are aggregated within blocks.

Properties

Property Name	Type	Description
transactionId	string	Unique identifier for the transaction assigned to the transaction by the Ledger System.
amount	Unsigned integer	The transaction amount to be deducted from the payer account and added to the receiver's account. The amount must be greater or equal to 0 and less than or equal to max.integer.
fee	Unsigned integer	The fee is taken from the payer account and added to the master account. The fee must be greater than or equal to 10.
note	string	An arbitrary string that may be up to 1024 characters in length.

Associations

Association Name	Type	Description
payer	Account	The Account issuing the transaction. The amount of the transaction and the transaction fee will be deducted from the payer's account balance.
receiver	Account	The amount of the transaction will be added to the balance of the receiver account.

Block

The Block aggregates groups of 10 transactions. Transactions should be added to blocks in the order that they are received. Prior to adding a transaction to a block, the transaction must be validated. Transactions that are invalid should be discarded. The block also contains an account balance map that reflects the balance of all accounts after all the transactions within the block have been applied. The account balance map should be copied from the previous block and updated to reflect the transactions in the current block. The block contains the hash of the previous block. It also contains the hash of itself.

Properties

Property Name	Type	Description
blockNumber	Unsigned integer	A sequentially incrementing block number assigned to the block. The first block or genesis block is assigned a block number of 1.
previousHash	string	The hash of the previous block in the blockchain. For the genesis block, this is empty. Use the Sha-256 algorithm and Merkle tree to compute the hash per the requirements.
hash	string	The hash of the current block is computed based on all properties and associations of the current Block except for this attribute. Use the Sha-256 algorithm and

		Merkle tree to compute the hash per the requirements.

Associations

Association Name	**Type**	**Description**
transactionList	Transaction	An ordered list of Transactions that are included in the current block. There should be exactly 10 transactions per block.
accountBalanceMap	Account	The full set of accounts managed by the Ledger. The account balances should reflect the account state after all transactions of the current block have been applied. Note that each Block has its own immutable copy of the accountBalanceMap.
previousBlock	Block	The previousBlock association references the preceding Block in the blockchain.

Ledger

The Ledger manages the Blocks of the blockchain. It also provides the API used by clients of the Ledger. The Ledger processes transaction processing requests, and also queries about the state of the Ledger, including Account balances, Transaction details, and Block details.

Properties

Property Name	**Type**	**Description**
name	string	Name of the ledger.
description	string	Ledger description.
seed	string	The Seed that is used as input to the hashing algorithm.

Associations

Association Name	**Type**	**Description**
genesisBlock	Block	The initial Block of the blockchain.

blockMap	map<uint,Block>	A map of block numbers and the associated Blocks.

Methods

Method Name	Signature	Description
createAccount	(accountId:string)	Create a new account, assign a unique identifier, and set the balance to 0. Return the account identifier.
processTransaction	(transaction:Transaction):transactionId:string	Process a transaction. Validate the Transaction and if valid, add the Transaction to the current Block and update the associated Account balances for the current Block. Return the assigned transaction id. If the transaction is not valid, throw a LedgerException.
getAccountBalance	(address:string):uint	Return the account balance for the Account with a given address based on the most recent completed block. If the Account does not exist, throw a LedgerException.
getAccountBalances	():map<string,uint>	Return the account balance map for the most recently completed block.
getBlock	(blockNumber:uint):Block	Return the Block for the given block number.
getTransaction	(transactionId:string):Transaction	Return the Transaction for the given transaction id.
validate	():void	Validate the current state of the blockchain. For each block, check the hash of the previous hash, make sure that the account balances total to the max value. Verify that each completed block has exactly 10 transactions.

LedgerException

The Ledger Exception is returned from the Ledger API methods in response to an error condition. The Ledger Exception captures the action that was attempted and the reason for the failure.

Properties

Property Name	Type	Description
action	string	Action that was performed (e.g., “submit transaction”)
reason	string	Reason for the exception (e.g. “insufficient funds in the payer account to cover the transaction amount and fee”).

CommandProcessor

The CommandProcessor is a utility class for feeding the Ledger a set of operations, using command syntax.

The command syntax specification follows:

Create Ledger

Create a new ledger with the given name, description, and seed value.

```
create-ledger <name> description <description> seed <seed>
```

Create Account

Create a new account with the given account id.

```
create-account <account-id>
```

Process Transaction

Process a new transaction. Return an error message if the transaction is invalid, otherwise output the transaction id.

```
process-transaction <transaction-id> amount <amount> fee <fee> note
<note> payer <account-address> receiver <account-address>
```

Get Account Balance

Output the account balance for the given account

```
get-account-balance <account-id>
```

Get Account Balances

Output the account balances for the most recently completed block.

```
get-account-balances
```

Get Bock

Output the details for the given block number.

```
get-block <block-number>
```

Get Transaction

Output the details of the given transaction id. Return an error if the transaction was not found in the Ledger.

```
get-transaction <transaction-id>
```

Validate

Validate the current state of the blockchain.

```
validate
```

Comment lines are denoted with a # in the first column.

Sample script:

See attachment `ledger.script`

Methods

Method Name	Signature	Description
processCommand	(command:string):void	Process a single command. The output of the command is formatted and displayed to stdout. Throw a CommandProcessorException on error.
processCommandFile	(commandFile:string):void	Process a set of commands provided within the given commandFile. Throw a CommandProcessorException on error.

CommandProcessorException

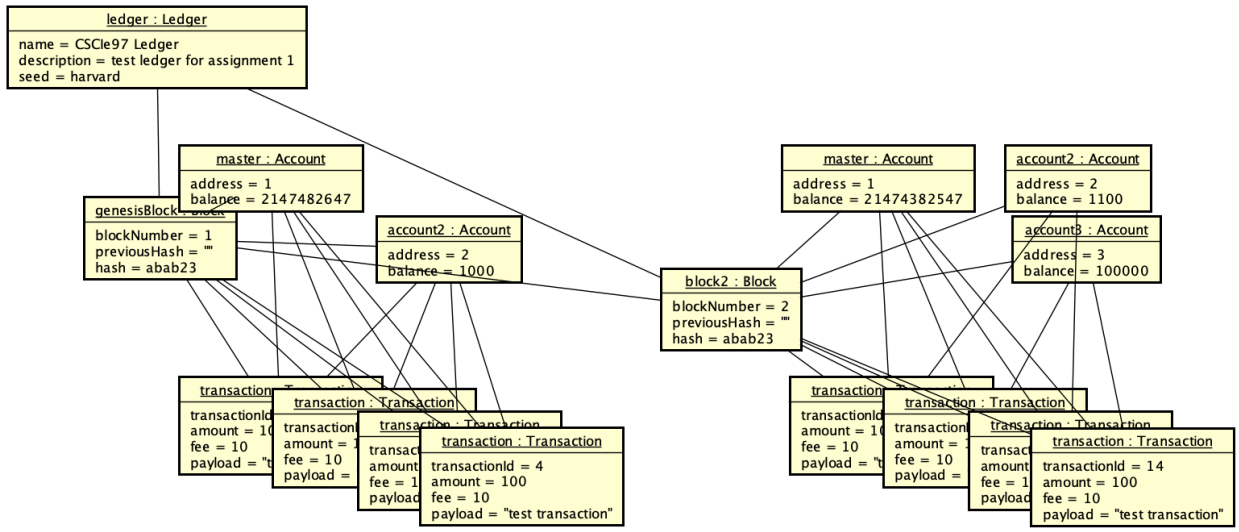
The CommandProcessorException is returned from the CommandProcessor methods in response to an error condition. The CommandProcessorException captures the command that was attempted and the reason for the failure. In the case where commands are read from a file, the line number of the command should be included in the exception.

Properties

Property Name	Type	Description
command	string	Command that was performed (e.g., “submit transaction”)
reason	string	Reason for the exception (e.g. “insufficient funds in the payer account to cover the transaction amount and fee”).
lineNumber	int	The line number of the command in the input file.

Implementation Details

The following instance diagram shows a Ledger with 2 blocks. Note that the transactions in each block are 4 rather than the expected 10.



Ledger Service UML Instance Diagram

The core component for the Ledger Service is the Ledger class. The Ledger class provides an API for interacting with the Ledger and implements the API methods that manage the Transactions, Accounts, and Blocks that make up the Ledger. The Transactions, Blocks, and Accounts are mostly stateful objects. The implementer may find that there are some useful utility methods on those objects, for example, the Block will likely have some methods to help with adding and retrieving Transactions and Accounts.

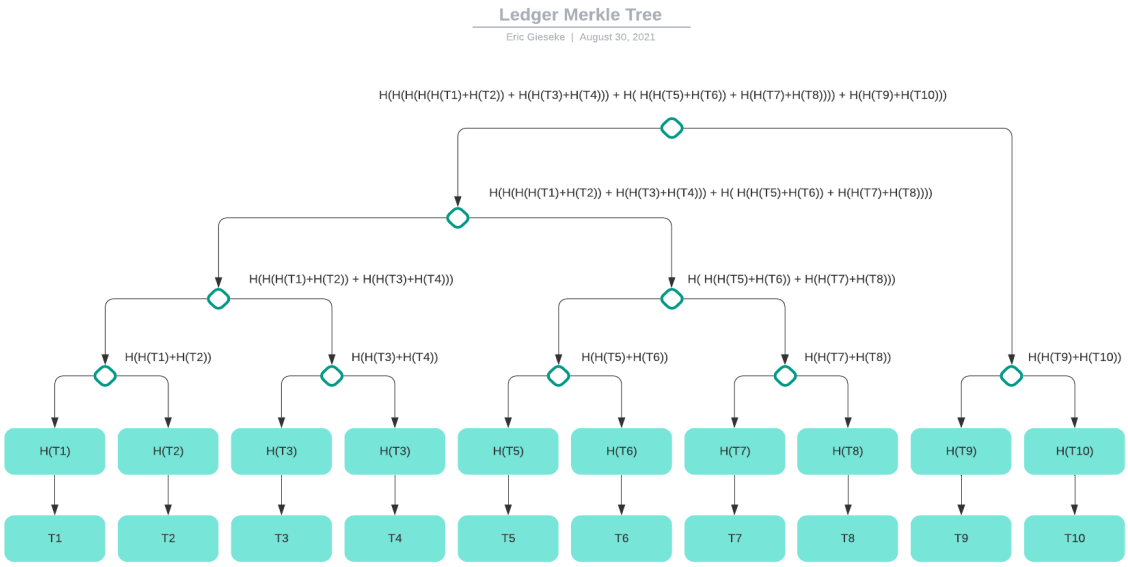
When creating new blocks, copy the account balances from the previous block, and augment those values as new transactions are processed. Each block should have a record of the account balances when the block was completed.

Since this is a blockchain application that will be dealing with monetary value, it is critical that the account balances be maintained correctly. Extra care should be taken to follow the guidelines for transaction processing. Account balances should never be less than 0 and the total of all account balances at the end of each block should always equal the starting value of the master account in the genesis block.

It is also critical that the blocks be linked correctly by including the hash of the previous block as a field of the current block. Computation of the hash should include the account balance table and the transactions within the block. This will prevent manipulation of either the account balances or the transactions in the future. Remember to include the Ledger Seed value when computing the hash.

Validate that account balances are correct. Make sure that each completed block has exactly 10 transactions. Attempts to process invalid transactions or create accounts that already exist should generate error messages (not stack traces).

The following diagram explains how to compute the merkle root for a set of transactions contained within a block.



Example computation of Merkle root

Testing

Implement a test driver class called TestDriver that implements a static main() method. The main() method should accept a single parameter, which is a command file. The main method will call the CommandProcessor.processCommandFile(file:string) method, passing in the name of the provided command file. The TestDriver class should be defined within the package “com.cscie97.ledger.test”.

A test command file will be provided, but you should create your own test file and process that. Include the test input and output with your submission.

Risks

Since this is a system dealing with monetary value, it will be subject to hackers who will attempt to undermine it. As soon as possible, transaction processing should be updated to require the signing of the transactions by the payer account.

The in-memory implementation leaves the system vulnerable to losing the state of the blockchain, transactions, and account balances. This should be corrected as soon as possible.

.