



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

*Distributed  
Computing*



# Bitcoin On Tendermint

Semester Thesis

Michael Bachmann

`michabac@student.ethz.ch`

Distributed Computing Group  
Computer Engineering and Networks Laboratory  
ETH Zürich

**Supervisors:**

Tejaswi Nadahalli

Prof. Dr. Roger Wattenhofer

June 29, 2020

# Acknowledgements

First I would like to thank my supervisor Tejaswi Nadahalli for his never-ending help, support and patience in the weekly zoom meetings and in his fast and elaborate replies to my e-mails despite the unusual coronavirus situation. Also I would like to thank Prof. Dr. Roger Wattenhofer and the DISCO group for this great opportunity in the first place.

# Abstract

Bitcoin on Tendermint is an application that implements a similar cryptocurrency like Bitcoin and it is written in Python3. It is built such that it runs on the application blockchain interface of the byzantine-fault tolerant consensus software named Tendermint. It contains a transaction parser, transaction validation, signature verification, a utxo set database, a transaction journal database and an application state to disc storage. This was a mainly hands-on project with the goal to mimic normal Bitcoin transactions, currency generation and multi-sig transactions. To test the entire system end to end a sample user is simulated with an integration test script. The script creates a variety of valid and a variety of invalid transactions, broadcasts them to Tendermint and includes certain assertions with the corresponding test conditions.

# Contents

<b>Acknowledgements</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>2</b>
2.1 Blockchains . . . . .	2
2.2 Bitcoin . . . . .	2
2.2.1 Bitcoin Consensus Mechanism Outline . . . . .	2
2.2.2 Bitcoin Concept . . . . .	3
2.2.3 Bitcoin Software Structure . . . . .	4
2.3 Tendermint . . . . .	4
2.3.1 Tendermint Structure . . . . .	4
2.3.2 Tendermint Validators . . . . .	4
2.3.3 Tendermint Consensus Mechanism Outline . . . . .	5
2.3.4 Tendermint ABCI . . . . .	5
2.3.5 Reproduce Bitcoin Logic on Tendermint . . . . .	6
<b>3 Implementation</b>	<b>8</b>
3.1 <code>init_chain()</code> . . . . .	8
3.2 <code>check_tx()</code> . . . . .	9
3.3 <code>begin_block()</code> . . . . .	9
3.4 <code>deliver_tx()</code> . . . . .	9
3.4.1 <code>validate_content_and_update_app_state()</code> . . . . .	9
3.5 <code>end_block()</code> . . . . .	11
3.6 <code>commit()</code> . . . . .	11
3.7 <code>info()</code> . . . . .	11

CONTENTS	iv
3.8 query() . . . . .	11
3.9 Integrity Check . . . . .	11
3.10 Consensus Based Applications . . . . .	12
3.11 Used Libraries . . . . .	12
3.12 Used Interface . . . . .	12
3.13 Used Tendermint Version . . . . .	12
<b>Bibliography</b>	<b>13</b>
<b>A Signature</b>	<b>A-1</b>
A.1 Non-trivial Signature Preimage . . . . .	A-1
A.2 Challenges in the Signature Verification . . . . .	A-1
A.3 Difference to the actual Bitcoin Signature Verification . . . . .	A-1
<b>B Conceptual Bitcoin Transaction Analogy</b>	<b>B-1</b>

# Introduction

---

The energy consumption of proof of work based cryptocurrencies worldwide has risen to a non-negligible amount over the last years due to the cryptocurrency boom. For one thing, as climate change is still a central issue in our society the demand for energy efficient alternatives to proof of work consensus technology is growing. For another, proof of work, like it says in the name, requires approximately 10 minutes of computational "work" every time before new information can be added to the distributed database. This means there is a lot of speed up potential. One approach to the energy problem is separating the business logic from the energy costly proof of work consensus software. The implementation of such an application on an already existing non proof of work consensus machine is a tiny but definitely interesting start to maybe one day contribute to an improvement of these concepts.

# Background

---

## 2.1 Blockchains

A blockchain is a data structure where the data is divided into an arbitrary amount of blocks such that if you merge the blocks back again, you get the entire data. Inside each block the hash digest of the previous block is stored. That way you always know which block was any block's predecessor. So in case you know the entire block set but do not know the block order you can in fact compute their original order. The first block is called the genesis block.

## 2.2 Bitcoin

### 2.2.1 Bitcoin Consensus Mechanism Outline

Bitcoin's [1] [2] consensus engine is based on blockchains. Although a blockchain is a useful tool in consensus technology it does not naturally guarantee data integrity or consensus amongst oppositional parties. A malicious party for example could easily modify the data in a certain block and recompute all the following blocks included with the correct hash digests. Therefore Bitcoin added a nonce field within the blockchain data such that one is able to influence or rather vary the block hash digest by varying the nonce value. This way namely one could agree that a party is only allowed to add a new block to the blockchain if the hash digest of the corresponding block fulfils some predefined conditions. When then a party wants to add a new block it can change the nonce value as many times as required until the hash digest is valid. Due to the properties of a cryptographic hash function the best approach to find a valid hash digest is brute forcing. These requirements to the hash digest can be adjusted in such a way that the computational work until at least one entity attained such a valid hash digest will on average take a desired amount of time. This way it's practically impossible to recompute all the following blocks after changing data in an intermediate block. This concept is called proof of work and is the one used by Bitcoin. The entities

that are trying to add new blocks are called miners.

### 2.2.2 Bitcoin Concept

Bitcoin is built and based on transactions. To understand the concept of Bitcoin as a cryptocurrency one first needs to know what a Bitcoin transaction is.

#### Bitcoin Transaction

A Bitcoin transaction consists of three main parts namely the transaction id, a set of transaction inputs and a set of transaction outputs. The transaction id is nothing else than the transaction data hashed twice using the hash function SHA256. The transaction id is unique to a transaction. A transaction input consists of a previous transaction output and a signature script. The previous transaction output is indicated by the transaction id and the transaction output index. An normal Bitcoin transaction output consists of a Bitcoin address and the amount you would like to send to that address.

#### Bitcoin as Currency

In Bitcoin to have money means to have one or multiple unspent transaction outputs more precisely to possess the private key to the public key of the Bitcoin address recorded within the unspent transaction output. The amount of currency the person possesses is equal to the value recorded in the unspent transaction output. When you want to spend the value in the unspent transaction output you can do this by creating a new transaction. The new transaction has to specify who the new owner is going to be by setting the correct receiver address in the transaction output. It also has to be signed as proof that the person spending the value in the unspent transaction output actually possessed it in the first place. A really important thing to understand is that once a transaction output is spent this specific transaction output will never be spendable again. I can't stress this enough. There will be no rest value that you can keep in this specific transaction output. The total value that you would like to spend in a transaction (all the indicated source transaction outputs as inputs), can however be split up into multiple new unspent transaction outputs. More precisely in a transaction a transaction input can be split into multiple transaction outputs. Also in a transaction multiple transaction inputs can be merged into fewer transaction outputs. With this whole concept you can return change to your own Bitcoin address again. It will just be a different unspent transaction output but it will be your money. If you want your transaction to officially be recorded and accepted by everyone, you have to ask a miner to add your transaction to a new block where it will be seen by everyone. To get a better intuition about this see Appendix B.



## Currency Generation in Bitcoin

In Bitcoin money gets created every time a miner adds a new block. They get rewarded with an unspent transaction output that they are allowed to include in the mined block. This unspent transaction output is wrapped inside a special type of transaction called the coinbase transaction. This unspent transaction output will have the value of a predefined miner reward plus the sum of all the miner fees of the in the block included transactions. The miner fees are simply the difference between all the transaction input values and the transaction output values of all the transactions in the block included. The first amount of money was created by the genesis block.

### 2.2.3 Bitcoin Software Structure

The Bitcoin software is composed of a consensus mechanism, a networking mechanism and the business logic. They are extremely coupled such that they are basically inseparable. This means you can hardly change one mechanism without changing, affecting or having to change another. Thus, you can't simply use the Bitcoin software as consensus engine for any type of application logic unfortunately. This would have been quite convenient.

## 2.3 Tendermint

### 2.3.1 Tendermint Structure

Tendermint [3] [4] consists of a consensus engine with a peer to peer network and simply an application interface called ABCI. ABCI stands for application blockchain interface. It is a clearly defined interface with a set of functions. Its communication is binary and based on Protocol Buffers. As any programming language can handle Protocol Buffers you can build an application on Tendermint in any language. So if you need a consensus based application, with Tendermint's structure you have the luxury of disregarding the consensus mechanism and the network and only focusing on the application itself. A thing to notice is that Tendermint only knows the execution states of the consensus mechanism and the network. The execution states of the application logic are only known to the application.

### 2.3.2 Tendermint Validators

What a miner in Bitcoin is, is called a validator in Tendermint so if a Tendermint node adds a new block it's called a validator. From now on data that is supposed to be added to the Tendermint blockchain will be referred to as a transaction.

When Tendermint gets initialized after installation the validator set is hard-coded into a file named `genesis.json`. This can however be changed as a part of the execution. Adding a new block in Tendermint is called validating a new block.

### 2.3.3 Tendermint Consensus Mechanism Outline

Tendermint uses a proof of stake based consensus mechanism regarding the voting power of all validators. The voting power of the validators is basically their stake. The higher their voting power the more often they are allowed to validate a new block. The voting power also is hard-coded into the `genesis.json` file and can also be changed as a part of the execution just like the validator set. Tendermint is byzantine-fault tolerant regarding the voting power of all validators as a new block needs to be approved of two third of all the voting power before being added to the blockchain.

### 2.3.4 Tendermint ABCI

#### **`init_chain()`**

This function gets called by Tendermint before the genesis block is created. It is mainly used to initialize application state variables.

#### **`check_tx()`**

This function is one of the two main functions of the application. It gets called whenever a transaction first arrives at the mempool. It decides if the transaction should be marked as a proposal transaction or removed from the mempool. It basically does the transaction parsing.

#### **`begin_block()`**

This function gets called every time before a new block gets validated. With this function the application can get information about the validator and do some computations and application state updates if required by the application functionality.

#### **`deliver_tx()`**

This function is the second one of the two main functions of the application. When a validator validates a new block and adds the proposal transactions to the block this function gets called multiple times in a loop until the block is full. It validates the content of the transaction by obeying the application logic. It

determines if the transaction will be marked as valid or as invalid in the new block. At this point it is important to mention that compared to Bitcoin where the business logic is used to order transactions into new blocks in Tendermint only `check_tx()` is used to filter transactions. All the transactions that pass the parsing are ordered into the blockchain no matter if they pass the application logic or not. They are merely tagged. This means in Bitcoin there will never be an invalid transaction inside an approved block but in Tendermint there most likely will. If this function gets called it will always be called in between `begin_block()` and `end_block()`. It is also responsible to update the application state.

### **end\_block()**

This function gets called every time a block, that is about to be validated, is full and ready to be committed. With this function the application can update the validator set and the voting power of the validator set. It can also do some computations and application state updates if required by the application functionality.

### **commit()**

This function always gets called after the `end_block()` function when a block is about to be committed. It should return the merkle root hash digest of the application state such that every application state can be associated with a merkle root hash digest.

### **info()**

This function returns information about the application state. It is used to synchronize Tendermint with the application during a handshake that happens on startup. [5]

### **query()**

This function gets called whenever a user queries the application state. It is supposed to return the implemented application state.

## **2.3.5 Reproduce Bitcoin Logic on Tendermint**

First it needs to be said that thanks to Tendermint no consensus algorithm neither any network needed to be implemented. What the application contains is transaction parsing, content validation, signature verification, merkle tree updating unspent transaction output set updating, ledger updating and data storage

updating. It can handle currency generation, normal Bitcoin transactions and multi-sig transactions. The currency generation transactions are quite different from the actual Bitcoin coinbase transactions. The former are not coupled to validators like coinbase transactions to miners but to a universal generation private key instead. They do however only get marked as valid if they possess no more than one input and one output just like a coinbase transaction. The application disregards transaction fees which could be implemented in a future improvement of the project. Except for multi-sig transactions it does not contain any advanced scripting abilities which could also be added. At last timelocks and SegWit transactions would pretty much make the application logic similar to the one of Bitcoin.

# Implementation

---

## 3.1 `init_chain()`

The path and empty files of the additional data to disc storage get created. They can be modified in the source code or the code could just be modified that way such that the path can be passed to the application as an argument from the terminal. Important to mention here is that the way the application stores data to the disc is simply a single file storage that gets overwritten every time additional data is stored. For one this is highly inefficient as the entire data has to be rewritten every time. For another, this is risky as important data could be lost in case the application crashed right before the data storage. In a future implementation this should be replaced either by an in-process database where the database is managed by an in the application included library that does this efficiently and integrally or by a full fledged database with an own process where your application connects to and receives replies.

In `init_chain()` also these application state variables are created:

- merkle tree (pymerkle MerkleTree)
- transaction counter (integer)
- latest block height (integer)
- merkle root of application state when latest block was committed (bytes object)
- validator address (string)
- unspent transaction output set (dictionary)

And last the empty dictionaries from the unspent transaction output set and the ledger are also already saved in the additional data storage.

## 3.2 `check_tx()`

In this function the transaction gets parsed and checked for a zero output value with a try/except block. The exception just returns code 1 to Tendermint which means that the parsing failed and the transaction will be removed from the mempool. The parsing is done using the `btcpy` library parser.

The `check_tx()` function could theoretically already validate the entire content of the transaction. This would however involve a loss in performance as the content would necessarily have to be validated again in the `deliver_tx()` function to ensure correct transaction validation. It probably depends on the application type if the price for having less transactions marked as invalid in the blockchain is worth the performance loss or not.

## 3.3 `begin_block()`

In this function the application reads out the validator address just in case that in a future implementation this would be needed.

## 3.4 `deliver_tx()`

In this function the application calls exactly one function namely

### 3.4.1 `validate_content_and_update_app_state()`

It's the main function of the business logic. It validates the content of the transactions and updates the application state variables and the data storage. It has a return value with an error code that gets caught in an if statement if the error code is not zero.

Within this one function first the generation transaction logic gets separated from the normal Bitcoin and multi-sig transaction logic with an if/else statement. In those blocks the content is validated. After the if/else statement, in case the transaction was accepted, the transaction is added to the ledger, the merkle tree is updated, the transaction count is incremented by one and the function returns code zero (valid) to `deliver_tx()`.

### Generation Transaction Logic

The transaction enters the generation transaction code block if at least one input contains the transaction id of 64 zeros. That's the tag of the fictional generation

transaction output. Otherwise it enters the other block. Within the generation block the application asserts there is only one input and one output in the transaction to make it mimic a Bitcoin coinbase transaction a bit more. This is implemented with a try/except block. Then, the generation transaction signature gets verified also with a try/except block and all the transaction outputs are added to the unspent transaction output set and to the data storage. The exceptions just return the corresponding error code to `deliver_tx()`.

### **Generation Transaction Signature Verification**

In a generation transaction the application simply verifies the single signature with the universal currency generation public key from the fictional transaction output. This key pair was temporarily put into the folder `btcpy_keypairs` and can be modified for advanced and future implementations. Some details about the signing process and the signature verification can be found in [Appendix A](#).

### **Normal Bitcoin and multi-sig Transaction Logic**

In this block within three try/except blocks the signature gets verified, the transaction gets inspected for sufficient source funds and also inspected for double spending of the source transaction outputs. The fund checking function also computes and returns the transaction fee in case this would be needed for a future implementation. The exceptions of this block just return the corresponding error code to `deliver_tx()`. After the try/except blocks the new transaction outputs are added to the unspent transaction output set and the spent source transaction outputs are removed therefrom. The same is done with the data storage of the unspent transaction output set.

### **Normal Bitcoin Transaction Signature Verification**

In an normal Bitcoin transaction the application verifies the signature in each transaction input with the public key of the corresponding source transaction output.

### **Multi-sig Transaction Signature Verification**

In a multi-sig transaction, for a single specific transaction input, the application tries to verify each contained signature in the input with all the public keys from the corresponding source transaction output. It also checks if the number of contained signatures equals the required amount of signatures specified in the source transaction output. For every signature the application has to find a matching public key to not fail.

### **add\_tx\_to\_ledger\_and\_to\_merkle\_tree()**

In this function the transaction is added to the ledger and the data storage of the ledger. Also the merkle tree is updated with the transaction data. For this the pymerkle function `merkletree.encryptRecord()` is used instead of `merkle-tree.update()` as according to the documentation this is recommended and more high-level. [6]

### **3.5 end\_block()**

In this function the application only increments the latest block height application state variable.

### **3.6 commit()**

In this function the application returns the current merkle root of the application state. The `merkletree.get_commitment()` function returns the same as the `merkletree.rootHash` attribute but when the tree is empty it returns `None` instead of raising an exception. [7] The application only returns the merkle root if the `get_commitment()` return value is not `None` otherwise the application returns no argument.

### **3.7 info()**

In this function the application returns the latest block height and the merkle root from when the latest block was committed. This is only returned if `init_chain()` had been called yet. Otherwise it simply returns zero and an empty bytes object.

### **3.8 query()**

In this function the application simply returns the unspent transaction output set dictionary json encoded as bytes object.

### **3.9 Integrity Check**

The integrity check first creates a whole bunch of valid transactions and also invalid ones with either incorrect transaction syntax or invalid transaction content. Then it broadcasts them to Tendermint in a specific order and queries the



blockchain before and after this process. The queried data is used to assert that either no block is added to the blockchain (failed `check_tx()`), a transaction is added but marked as invalid (failed `deliver_tx()`) or a transaction is added and marked as valid (successful `deliver_tx()`). It has to be mentioned that this integrity check will not work for a network with more than one validator node and one user as despite of a transaction failing the `check_tx()` another validator could still add another block. Also if another user sends transactions the hard-coded block height used to query the blockchain could query the incorrect block. This would have to be considered in a future improvement of the project.

### 3.10 Consensus Based Applications

Implementing consensus based software should always be very carefully thought through before actually initiating the software. Because one needs to be aware of the fact that once any data has been added to the blockchain it is impossible to change consensus sensitive syntax of the protocol without taking the whole system down. Shutting down the whole system can be impossible or with grave consequences depending on the functionality the application is used for. So a bug will possibly have to be carried over forever.

### 3.11 Used Libraries

- `btcpy` (transaction creation, transaction signing and transaction parsing) [8]
- `ecdsa` (manual signature verification of the transaction) [9]
- `pymerkle` (merkle tree creation and updating) [10]
- `google.protobuf` (read out validator information from the `RequestBeginBlock` message) [11]

### 3.12 Used Interface

Dave Bryson's Python3 interface for Tendermint v0.32.6 [12]

### 3.13 Used Tendermint Version

Tendermint v0.32.6 [13]

# Bibliography

- [1] “Bitcoin whitepaper,” <https://bitcoin.org/bitcoin.pdf>, accessed: 2020-06-28.
- [2] “Bitcoin developer guide,” <https://developer.bitcoin.org/devguide/>, accessed: 2020-06-28.
- [3] “Tendermint whitepaper,” <https://tendermint.com/static/docs/tendermint.pdf>, accessed: 2020-06-28.
- [4] “Tendermint explained on Tendermint homepage,” <https://docs.tendermint.com/master/introduction/what-is-tendermint.html>, accessed: 2020-06-28.
- [5] “Tendermint info() function,” <https://docs.tendermint.com/master/spec/abci/abci.html#info>, accessed: 2020-06-24.
- [6] “Pymerkle encryptrecord() function,” <https://pymerkle.readthedocs.io/en/latest/encryption.html#single-record-encryption>, accessed: 2020-06-27.
- [7] “Pymerkle get\_commitment() function,” <https://pymerkle.readthedocs.io/en/latest/merkle-proofs.html>, accessed: 2020-06-26.
- [8] “Btcpy library,” <https://github.com/chainside/btcpy>, accessed: 2020-06-28.
- [9] “Ecdsa library,” <https://github.com/warner/python-ecdsa>, accessed: 2020-06-28.
- [10] “Pymerkle library,” <https://github.com/fmerg/pymerkle/blob/master/docs/source/index.rst>, accessed: 2020-06-28.
- [11] “Protocol buffer library,” <https://github.com/protocolbuffers/protobuf>, accessed: 2020-06-28.
- [12] “Python3 interface for tendermint,” <https://github.com/davebryson/py-abci>, accessed: 2020-06-28.
- [13] “Tendermint version v0.32.6,” <https://github.com/tendermint/tendermint/releases/tag/v0.32.6>, accessed: 2020-06-28.

# Signature

---

## A.1 Non-trivial Signature Preimage

To create the signature preimage of a transaction whose input is about to be verified, the signature scripts of all inputs, except for that particular one, are replaced by empty signature scripts. The one that was left out is replaced by the public key script of the corresponding source transaction output. Even though in the application the signature preimage to verify the signature is created by the `btcpy` library, it could still be beneficial to know how it is actually done.

## A.2 Challenges in the Signature Verification

One needs to know that the `btcpy` library and the `ecdsa` library are not naturally compatible. Bitcoin encodes the signature stored in the transaction in the DER format, not in the string format like the `ecdsa` library does. This needs to be considered when verifying any signature. Also Bitcoin subtracts the `ecdsa S` parameter from the `ecdsa order` parameter and uses this computed value as the new `ecdsa S` parameter value when encoding it back into DER format. If this is ignored the signature verification will fail every time.

## A.3 Difference to the actual Bitcoin Signature Verification

Compared to the application that uses two simple loops for signature verification Bitcoin uses a rather complex low-level stack protocol for that. This is another area where the application has some potential for improvement of efficiency.

# Conceptional Bitcoin Transaction Analogy

---

It is not so easy to get a good grasp of the intuition behind Bitcoin transactions as a currency. Therefor one might want to think of a conceptional analogy of it. This might sound rather funny or odd to some but may even be helpful to others. Let's start the analogy by picturing multiple bowls. Each bowl has an arbitrary amount of gold coins inside. Each gold coin has an arbitrary mass and the value of the gold coin equals its mass. Now there are going to be some rules to this scenario. There are going to be rounds. Each round consists of two phases. In each round the value of arbitrary many bowls can be redistributed as follows. In phase one of each round a blacksmith melts any arbitrary set of gold coins of all the bowls together. Then in phase two of each round the blacksmith forms an arbitrary amount of new gold coins each with an arbitrary mass (using the melted gold mass from phase one) and redistributes them arbitrarily into any of the bowls he wishes. He also writes down the changes that he makes, really detailed, in some sort of journal after any few rounds. For each round he writes down the gold coins he took for melting and writes down where he added new gold coins and the new gold coin's masses. The blacksmith will steal a tiny bit of the melted gold in every round for his trouble. Now think of the gold coins in the bowls as only the *unspent* transaction outputs and think of their mass as the value inside the corresponding unspent transaction output. Think of the rounds as transactions. Think of the bowls as public keys (owners). Think of the journal as the blockchain and think of the stolen bit in every round as the transaction fee for the miners. You now have a similar concept to Bitcoin.