**The basic concept**

Blockchain is a distributed database that maintains a continuously growing list of ordered records called blocks secured from tampering (fraud) and revision.

**NaiveChain**

A super-simple blockchain. The NaiveChain does not have a “mining” algorithm (POS or POW) so it cannot used in a public network. It nonetheless implements the basic features for a functioning blockchain.

Source: <https://medium.com/@lhartikk/a-blockchain-in-200-lines-of-code-963cc1cc0e54>

I am going to re-write the code in Python.

**Key concepts of NaiveChain**

The basic functionalities of blockchain:

* A defined block and blockchain structure
* Method to add new blocks to the blockchain with arbitrary data
* Blockchain nodes that communicate and sync the blockchain with other nodes
* Use Websockets to communicate with other nodes (P2P)
* HTTP interface to control the node

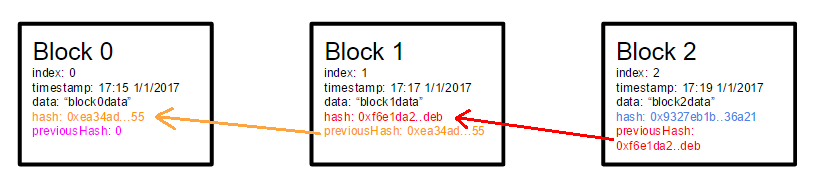
**Block structure**

The most necessary: index, timestamp, data, hash and previous hash.

* **Index**: the height of the block in the blockchain
* **Data**: any data is included in the block
* **Timestamp**: creation time of the block
* **previousHash**: a reference to the hash of the previous block
* **Hash**: a sha256 generated from all above contents of the block



For example:



The hash of the previous block must be found in the block to preserve the chain integrity

The code of class block (block.py):

|  |
| --- |
| class Block(object):  def \_\_init\_\_(self, index, sHash, previousHash, timestamp, data):  self.index = index  self.sHash = sHash  self.previousHash = previousHash  self.timestamp = timestamp  self.data = data |

**Block hash**

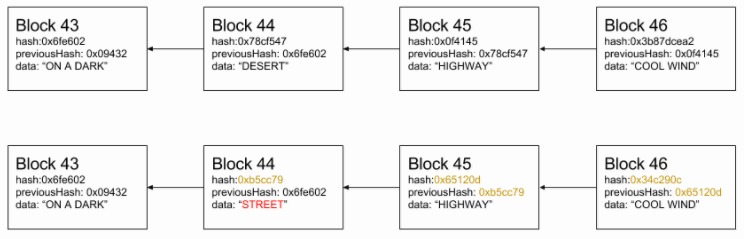
The block needs to be hashed to keep the integrity of the data. A SHA-256 is taken over the content of the block (index, previous hash, timestamp, and data). This means that if anything in the block changes, the original hash is no longer valid.

The hash code (blockchain.py):

|  |
| --- |
| import hashlib  # Block hash  def calculateHash(self, index, previousHash, timestamp, data):  sha256 = hashlib.sha256()  # TypeError: update() argument 1 must be string or buffer, not int or datetime  sha256.update(str(index))  sha256.update(previousHash)  #sha256.update(str(timestamp))  sha256.update(data)  return sha256.hexdigest()  def calculateHashForBlock(self, blockIn):  return self.calculateHash(blockIn.index, blockIn.previousHash, blockIn.timestamp, blockIn.data) |

(\*) This hash has nothing to do with “mining”, since there is no Proof of Work (POW) problem to solve. We use block hashes to preserve integrity of the block and to explicitly reference the previous block. A block can’t be modified without changing the hash of every consecutive block.

For example, if the data in block 44 is changed from ‘DESERT’ to ‘STREET’, all hashes of the consecutive blocks must be changed.



**Storing the blocks**

An array or a vector is used to store the Blockchain. The first block of the Blockchain is always a so-called “genesis-block”, which is hard coded.

The code (blockchain.py):

|  |
| --- |
| from block import \*  class BlockChain(object):  # Storing the blockchain in a list  lChain = []  def \_\_init\_\_(self):  lChain = [] |

**Genesis block**

Genesis block is the first block in the blockchain. It is the only block that has no previousHash.

The code (blockchain.py):

|  |
| --- |
| # Genesis block  def createGenesisBlock(self):  # sha256('Tung') = 'f97a02ba06587422d685927f0245114523b55578331b5eaa0c3d7f7ebb77fce0'  genesisBlock = Block(0, 'f97a02ba06587422d685927f0245114523b55578331b5eaa0c3d7f7ebb77fce0', '', datetime.now(), "Tung's genesis block")  self.lChain.append(genesisBlock) |

**Generating a block**

To generate a block we must know the hash of the previous block and create the rest of the required content (index, data, timestamp, and hash). Block data is something (e.g. financial transactions) that is provided by the end-user.

The code (blockchain.py):

|  |
| --- |
| def getLastestBlock(self):  return self.lChain[len(self.lChain) - 1]  # Generating a block  def generateNextBlock(self, nextBlockData):  previousBlock = self.getLastestBlock()  nextIndex = previousBlock.index + 1  nextTimestamp = datetime.now()  nextHash = self.calculateHash(nextIndex, previousBlock.sHash, nextTimestamp, nextBlockData)  newBlock = Block(nextIndex, nextHash, previousBlock.sHash, nextTimestamp, nextBlockData)  return newBlock  # Adding new block into chain  def addNewBlock(self, nextBlockData):  self.lChain.append(self.generateNextBlock(nextBlockData)) |

**Validating the integrity of blocks**

At any given time, we must be able to validate if a block or a chain of blocks are valid in terms of integrity. This is true especially when we receive new blocks from other nodes and must decide whether to accept them or not.

For a block to be valid the following must apply:

* The index of the block must be one number larger than the previous
* The previousHash of the block match the hash of the previous block
* The hash of the block itself must be valid

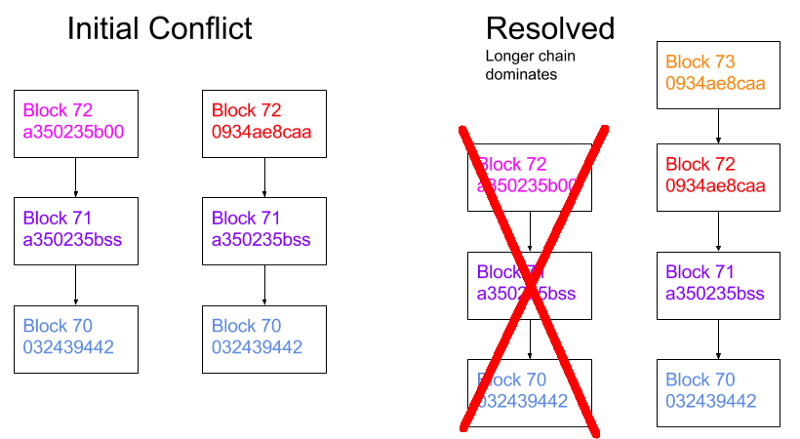
The code (blockchain.py):

|  |
| --- |
| # Validating the integrity of blocks  def isValidNewBlock(self, newBlock, previousBlock):  if (previousBlock.index + 1 != newBlock.index):  print 'invalid index'  return False  elif (previousBlock.sHash != newBlock.previousHash):  print 'invalid previousHash'  return False  elif (self.calculateHashForBlock(newBlock) != newBlock.sHash):  print 'invalid hash value'  return False  return True  # Validating the structure of the block, so that malformed content sent by a peer won't crash our node  def isValidBlockStructure(self, blockIn):  # Type of index should be int  # Type of sHash should be string  # Type of previousHash should be string  # Type of timestamp should be timedate  # Type of data should be string  return True  # Checking for valid block chains  def isValidBlockChain(self, lChainIn):  for i in range(1, len(lChainIn)):  # 1st check if valid block structure  # ...  # 2nd check if valid new block  if (self.isValidNewBlock(lChainIn[i], lChainIn[i - 1]) == False):  print 'Block ' + str(i) + ' is NOT valid!!!'  return False  print 'Block Chain is valid'  return True |

**Choosing the longest chain**

There should always be only one explicit set of blocks in the chain at a given time.

In case of conflicts (e.g. two nodes both generate block number 72) we choose the chain that has the longest number of blocks.



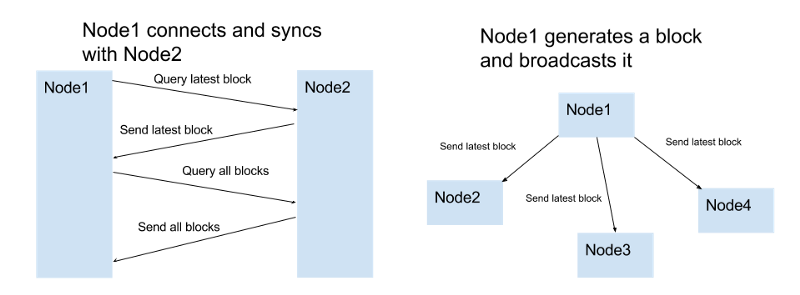
The code (blockchain.py):

|  |
| --- |
| # Choosing the longest chain  def replaceChain(self, newChain):  if (isValidBlockChain(newChain) == True & len(newChain) > len(lChain)):  print 'Received blockchain is valid. Replacing current blockchain.'  lChain = newChain  # broadcastLatest()  else:  print 'Received BlockChain invalid' |

**Communicating with other nodes**

An essential parts of node is to share and sync the blockchain with other nodes. The following rules are used to keep the network in sync.

* When a node generates a new block, it broadcasts it to the network
* When a node connects to a new peer it queries for the latest block
* When a node encounters a block that has an index larger than the current known block, it either adds the block to its current chain or queries for the full blockchain.



We use the Websockets for the P2P communication. The active sockets for each node are stored in the socket list. No automatic peer discovery is used. The location (Websocket URLs) of peers must be manually added.

**Controlling the node**

The user must be able to control the node in some way. This is done by setting up a HTTP server. The user is able to interact with the node in the following ways:

* List all blocks
* Create a new block with a content given by the user
* List or add peers

The code:

TBD

**Architecture**

It should be noted that the node actually exposes two web servers (reasons???). One for the user to control the node (HTTP server) and one for P2P communication between the nodes (Websocket HTTP server)



Testing the code

Main.py

|  |
| --- |
| from blockchain import \*  def main():  print "CREATING A CHAIN OF BLOCKS..."  bChain = BlockChain()  # Generate block 0  bChain.createGenesisBlock()  # Generate block 1  blockData1 = 'Satoshi sent Tung 100000btc; '  bChain.addNewBlock(blockData1)  # Generate block 2  blockData2 = 'Tung sent his mom 1000btc; ' + 'Tung sent his sister 10000btc; '  bChain.addNewBlock(blockData2)  bChain.isValidBlockChain(bChain.lChain)  for i in range(0, len(bChain.lChain)):  bChain.printBlockInfo(bChain.lChain[i]) |

Result:

