Coniunctum

A Peer-To-Peer Python Blockchain and Cryptocurrency

Alain Magazin

March 20, 2021

Decentralization is one of the main ideas circulating on the Internet these last few years. As bitcoin's price skyrockets, learning the basics of blockchain technology is a must for any aspiring programmer or software developer. Coniunctum is a simple blockchain project fully built in Python that aims to open up blockchain developing to newcomers in the developing world.

Basic vocabulary and overview

The basic concept of this cryptocurrency is fairly simple, and very similar to Bitcoin's protocol. The first version will be based on standard proof-of-work and will present basic solutions to problems encountered by cryptocurrencies. These solutions are not intended for large scale use and therefore can be updated along the way. For example, we will eventually update the blockchain protocol to some sort of proof-of-stake.

When a new client is installed, it is linked with a "source node", the node from where the new client was downloaded, it's from this node that the newly installed client will fetch blockchain data or the entire blockchain (depending on the network role chosen by the new user). The network of users will consist of three user types. The first type is the wallet holder, which represents the simplest model of use for this cryptocurrency. Wallet holders interact only with their source node so they don't have to download the entire blockchain. In this case trust in the source node is very important as a malicious node could trick its users into a wrong blockchain. Wallet holders broadcast their transactions to their source node. Next, we have nodes we refer to as validators (although this term may have different meanings in other blockchains). Their role is to listen for new transactions, verify them and send them to miners. They also create a unique installer of the Coniunctum Client that they can share, it is unique in the sense that users that install the client from this particular installer will have this node as their source node. And finally, miners, they listen for verifed transactions and add them to the block they are currently mining. A single user can be both a miner and a node and we refer to these users as full nodes (again, only here). Every miner / validator / full node is connected to every other miner / validator / full node in the network (not adapted for a large scale, we might reduce to a few connections) so that everyone can exchange transctions or block data. Wallet holders only connect to their source node, they don't need to know about everything going on in the network. Okay, we are now ready to dive more deeply into this blockchain...

Listening to and verifying transactions

- Public key, private key and wallet address

  Each user of the network needs some sort of identification. For this, each user will be granted a public key / private key pair, with a wallet address. The key pair will be automatically generated and the wallet address will be some hash of the public key (to facilitate wallet recovery). The private key will be stored in an encrypted file and the user will prompted to enter a password for this encryption while setting up the node/wallet.

- Digital signatures

  The first layer of security we need is a way of verifying wether the transaction really was created by the sender. Therefore we introduce digital signatures. Here we will not elaborate on the way that digital signatures work. It suffices to say that basically every transaction will have an unforgeable signature that can be verified easily with sender's public key. This ensures that transactions can only be created by the sender. To ensure that every transaction requires a different signature we add a unique ID to each transaction.

- Broadcasting new transactions to validators

  Transactions need to be verified by multiple nodes and transmitted to validators to be added to blocks. To do this in Python we use a basic socket server and a client. Python offers great tools for socket programming. The main idea is that every node will run a server and client, the client will send data when needed and the server will listen for data coming from other users. For this we use an unencrypted TCP protocol. Unencrypted because block information will eventually be made public on the blockchain. We may add some encryption later on.

- Verifying new transactions

  When a validator reveives transaction data, it'll contain the following information :

  - Sender's public address
  - Sender's wallet address
  - Receiver's wallet address
  - Unique transaction ID
  - The signature
  - The amount sent
  - The timestamp
  - Hashed transaction information

  In order to verify if the transaction is valid, the node needs to verify the signature, that the Tx (Tx = Transaction) ID is unique and that the sender has the right amount of coins to send. If the transaction gets validated it is automatically sent to miners via the TCP client. On the other hand if the transaction is not valid, the node's ip that the Tx came from will be logged, and nothing will be sent to miners.

Adding blocks to the blockchain

- The mining process

  When a miner receives transaction data either from it own node or from other validators, it will re-verify it with the same protocol as described above to prevent malicious nodes from adding malicious transactions to blocks. When the verification is completed, the miner stores the transaction in a local file. All this while searching for the proof of work for the next block. The handling of new transactions is done by the TCP server. The miner has to solve the following equation to find the proof :

  $$sha256(x^2 + PH^2) = "0" * diff... \quad (1)$$

  Where x is the proof the miner is looking for, PH is the hash of the previous block and difficulty is the integer telling how many zeros we want the hash of (1) to start with. The miner just tries different values of x starting from x = 1, as the sha256() function cannot be reversed. This should last for about 10 minutes for an average computer. When the miner finds the PoW, he creates a new block structure :

  - Block index
  - Miner's wallet address
  - Timestamp
  - Proof of Work
  - Hash of the previous block
  - Transctions list

  The transactions list will be filled with the transactions from the local Tx file and the miner will add a special transaction that rewards him a certain amount of Coniunctum coins.

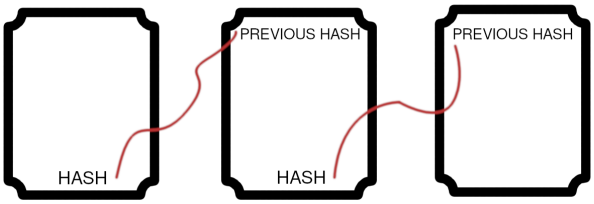
- Broadcasting new blocks

  After creating a new block, the miner needs to broadcast it to the rest of the network. Here again, the TCP client comes in to play - indeed, it's through the same connection used by transactions that the new block will be sent. When nodes receive a new block they proceed to a two-step verification. The first step is to verify the proof of the block and to verify that the previous hash matches. If this step succeeds, the node moves on to step two where it verifies all the transactions contained in the block with the same protocol as described before for new transactions.

  If the block is succesfully verified, the node adds it to it's blockchain and sends it to the rest of the network. Nodes trust the longest blockchain and if they receive a block they already have they simply ignore it. This verification process should prevent malicious transactions from entering the blockchain unless hackers control over 51% of the network.

- Syncing the blockchain

  On each restart, the client needs to sync with the blockchain of the network. To do that we scan the network for online nodes, and from those nodes, the client gets the blockchain. If there are conflicting chains, it chooses the one used by the majority of nodes. When the starting node chooses a blockchain to use it verifies it's integrity (i.e verifying that hashes are well linked) as described in the following scheme :



  If this verification succeeds the node downloads the blockchain and replaces it's current one.

- Final thoughts on this mining process

  By implementing this mining protocol we hope to solve the main challenges of a decentralized trust system. Blocks that are added to the blockchain cannot be changed as it would change the hash of the following block, that's why it's called a blockchain. If someone wants to hack this blockchain it would require more than 51% of the computational power of the network since nodes trust the longest blockchain and the blockchain used by the majority of the network.

  All the previously described verification should ensure that no fraudulus transaction gets validated.

Network organization

- Node identification

  To enter the network as a node or full node, several settings are mandatory. As we identify each node by its public IP address, the user has to set up port-forwarding for both the TCP server and the web app. The web app listens on port 56230 and the TCP server listens on port 5050. For the time being those settings are fixed but we might enable changing the port of the web app in the future.

- Connecting to other nodes

  If we assume a small sized network, a single node can be connected to every other node in the network. This offers faster data broadcasting but is limited to small networks. For a larger network, connecting to only a few other nodes would be suffecient if we connect to the right nodes that allow us to acces the whole network. The goal is to avoid, as often as possible, parts of the network linked to only one or two nodes. This creates the risk of a "network fork" in case the few linking nodes go offline. Although we recommend node operators to stay online as long as possible, we cannot prevent a node from going offline eventually.

- Network security

  All these connection requirements can be potential breaches in the node's computer. The web app only accepts GET http requests so it should not be a concern. The TCP server however can be a huge security breach as it collects data sent by other users. Research needs to be done on wether some sort of injection is possible, but for now we assume that if not-intended data gets sent it just gets rejected by the TCP handler.

  The data transmitted is, for now, not encrypted. This means attackers can read the data sent by nodes, though this is not a critical problem as transaction and block data is supposed to be published on the public blockchain. We might add encryption for educational purpose (or if we realise that, for some reason, it is absolutely necessary) later on.

Final thoughts

This is what an oversimplfied cryptcurrency/blockchain system would look like, in my opinion. As we continue developing this project, we should upgrade it, inspired by other blockchains or, perhaps, with our own new concepts. These upgrades might include a Proof-of-Stake consensus protocol (inspired by the Cardano project), the addition of smart contracts, native assets, etc. - basically, everything a modern blockchain has to offer. This project being coded in Python and not in a more performing language (like C++ or Haskell) will restrain it to a non-commercial use, but it would offer a way for new developers to understand blockchain technology without the "pain" of a low-level programming language.

I am myself a beginner in programming and in blockchain technology so this project is potentially full of security breaches and conceptional errors, so feel free to contact me if you have any suggestions.

- Contacts :

  Email : airwaks98701@protonmail.ch

- Donations :

  Bitcoin BTC : 19LkpNaT34UrLzU6a1X9UPFEjbD8pK7tBw

  Cardano ADA : addr1qxhcw33f7j9tppxr2dpfpnrz76laddly4pahks5pcpp5yzl7wqt86 frzq7ztcupgzwrt5f9w97v3t87v4rl6uxztkjuqwws8e9

References

[1] *Bitcoin: A Peer-to-Peer Electronic Cash System*, Satoshi Nakamoto, 2009.

[2] *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol*, Aggelos Kiayias, Alexander Russell, Bernardo David, Roman Oliynykov, 2019