Dan Stevens (djs447) Database System Engineering
Final Project Report 5/5/2024

Building a Blockchain Ledger

I. Introduction

The objective of this project is to build a blockchain ledger and demonstrate its basic functionality. The concept of a blockchain is important because it offers a decentralized method for storing all kinds of data using a shared, immutable ledger. A blockchain offers many of the same benefits as a traditional database, however there are some key differences that provide both strong advantages and unique challenges. Some of the benefits of employing blockchain include enhanced security through cryptographic hashing of each entry, advanced privacy thanks to no central governing body requiring oversight, and improved fault tolerance due to every participating node keeping a record of the data and removing a single point of failure from the equation. In some implementations however these advantages may appear to be weaknesses as some organizations would place greater value in the ability to modify data after it's been entered or the safety blanket offered by a central governing body. For this reason, implementing blockchain technology is not a one-size-fits-all solution, however this project will walk through some of the blockchain features which make it such a unique and promising form of data storage.

II. Background

For reference material I relied heavily on Satoshi Nakamoto's whitepaper on Bitcoin, titled "Bitcoin: a Peer-to-Peer Electronic Cash System", which details defining factors on how blockchain technology may be implemented to support the development of cryptocurrency [1]. For this project I did not want to limit the implementation of blockchain to Bitcoin or any other form of cryptocurrency, however this paper provided a solid foundation on what defines a blockchain and how it can be used to store information. Using this whitepaper, I started by writing a requirements document to highlight what functionality would be required to make this blockchain work as needed. These requirements were used to define the contents of a block, what determines the validity of a block / the blockchain overall, how new blocks would be added, and many other areas pertinent to the implementation overall.

Specifics of the data structures and their contents which define this blockchain can be found in the following section, and after that details are provided on how the blockchain functionality will be demonstrated to a larger audience. For the hashing algorithm, SHA-256 was used due to the importance placed on it in the whitepaper as well as the ease of implementation through Python libraries. If a less effective hashing algorithm had been used I would have liked to demonstrate how a simpler blockchain might be tricked into accepting altered or malicious data, however SHA-256 is much harder to manipulate. For the mining, each block stores data primarily in the form of a string. Functions have been included for both accepting user input data and for generating arbitrary data

automatically. In a cryptocurrency system, the data contained in each block might be expected to be some form of transaction, i.e., 'Person A transferred X coins from their wallet to Person B'. The way this has been implemented, since it is not tied to any existing or imaginary cryptocurrency, this data is limited to string transactions. While not tied directly to monetary values, this is still useful as it provides an immutable ledger for participating members to store data which cannot be altered.

The proof-of-work concept for mining works by defining an expected number of leading zeroes for each block's hash value within the block chain, and then regenerating the hash value until this leading number of zeroes is achieved. For each hash generation, a value within the block called the nonce is incremented by one to provide some variation in the returned hash value. Increasing these number of zeroes exponentially increases the difficulty of mining a block, and in order to be considered a valid block the block header, comprised of a set of data within the block, must satisfy this concept when hashed. For blockchains like Bitcoin, the difficulty of mining a block is adjusted on a regular basis to keep the difficulty relatively consistent over time. For this implementation the difficulty is managed by the number of zeroes defined for the blockchain and this is left static for all implementations of the blockchain object.

For this project I created the node, blockchain, and block objects to demonstrate a limited portion of how a blockchain functions. A user can create their own node which will retrieve a copy of the blockchain from another node on the network, or create a new blockchain if it does not detect any partners. They can then add new partners, mine new blocks, and share their version of the blockchain with all nodes within the network so that consistency is maintained for the data. In a deployed system this would all be done automatically, however this would make it hard to actually demonstrate individual functionality. For this reason, I have broken the key functions of a blockchain into individual steps so that they can be executed one-by-one and demonstrated to the user. The advantage of this is that the output of each step can be viewed independently.

III. Blockchain Structures

To make this implementation easier, I've split up the relevant blockchain elements into distinct structures which each perform vital functions to make the implementation work. In this section I'll walk through the different objects and the functions that they use.

A. Node

Contents

This was the last object created for the project, the node object represents the individual contributors to the blockchain ledger. Every computer which maintains a copy of the blockchain is an active node and in order to emphasize the decentralized nature of a blockchain new nodes can freely join the network as long as they maintain the chain. Each node is responsible for mining new blocks as needed, sharing the current status of the blockchain with its peers, and maintaining its own copy of the

blockchain. In the event there are any discrepancies between the blockchain versions maintained by two or more nodes within the network, there needs to be a protocol to resolve these discrepancies. This protocol, albeit simplified, is described in the *consensus.py* functionality and later within this paper.

- node_id: This is an integer identifier for the node used to label and track the different nodes participating in the blockchain network. This is set when the node is initialized and is incremented as necessary when new nodes join the network. In a system which is actually deployed, this would potentially be linked to the specific hardware or address of the user operating that node.
- peers: Each node maintains a list of the peers it is tracking on the network as this shows the other nodes which it is directly connected to. Since this is not a deployed system there are currently no limits to the number of peers a node should maintain or a process for sharing peers across nodes, however the next step in this implementation would be to have transactions and blocks sent directly from a peer to others over a network by their address.
- blockchain: This is the version of blockchain actually maintained by a specific node. Further information on the blockchain object can be found later in this paper, however for now the main thing to know is that at any given moment it is possible that not all nodes have the same version of the blockchain. These differences may be due to discrepancies on who mined a specific block first and delays in broadcasting a block across the network so there needs to be some way for all nodes to reach a consensus on the real, valid blockchain.

Supporting Functions

In this implementation, the node object makes use of the following supporting functions to complete the required functionality for the blockchain to work as expected.

- add_peer(self, peer): This function allows a node to update the list of peers it is maintaining with a new node. The peer in the input for this function refers to a node object, the default status of the peers list is empty however it is possible to pass peers on initialization as well.
- receive_chain(self, chain): In this function the node waits to receive a copy of the chain from a peer and if this chain is both valid and longer than the existing copy of the blockchain it is maintaining it will replace the version it has with the newer, better version. It will then save a copy of this chain locally as well.
- broadcast_chain(self): This function has the node send a copy of the blockchain it is maintaining to every node within its list of peers. All of these nodes are expected to run the receive_chain() function so that they can check if their version should be updated.
- block_save(self, block): Specific to this implementation, this function saves a copy of the block locally to the node in the form of a file named ‘000000X.json’, where X is the index of the block within the chain. The contents of the block are then entered into the JSON file so that if the node ever goes offline it can pick

back up where it left off, however if it is connected to a larger network it will need to retrieve the longest valid chain from other active nodes when it comes back online.

- chain_save(self, chain): This function extends the block_save() functionality introduced above. The difference here is that for this save it will save a copy of each element within the blockchain locally instead of only looking at individual blocks. The same naming convention is followed and the same directory is used.
- sync(self): This function is executed whenever a new node is initialized or when a chain is being retrieved from local memory. It works by first creating an empty blockchain and then checking to see if a file directory "chaindataX" exists, where X is the node_id value for the specific node running this function. If it does not, it creates this directory, finds the best version of the blockchain available on the network, and creates a copy locally to this directory, as well as saving the best version to that specific node. This allows new nodes to join in and still receive the latest and greatest version of the blockchain that it should be maintaining. If this is the first node to join the network, it will instead initialize the blockchain and create a copy of it locally.

  If the directory does exist, then the sync() function will instead read the block files from this directory and use this to update the blockchain being maintained by the node. This functionality allows a node which may have previously had a copy of the blockchain but did not receive an updated version (by being disconnected, for example) to jump back in and participate in the network again.

B. Blockchain

Contents

The blockchain object is the backbone of this ledger as it contains a sequence of block objects that cannot be changed once they are recorded without having to re-create the entire blockchain. This works because each block within the chain is hashed using data from previous blocks and contains references to these blocks such that any node participating in the blockchain's network can validate the chain.

- blocks: This is a list of blocks within the blockchain, as more blocks are mined they are added to the list in order. Fairly straightforward, this list of blocks can be iterated through to confirm the validity of the blockchain by checking both the validity of each block itself as well as the validity between the relationships of neighboring blocks.
- zeroes: For this implementation the zeroes variable refers to the number of leading zeroes that need to be present in the hash value for a block in order for it to satisfy the proof-of-work concept. Every time a block is mined, it will check to see if the number of leading zeroes in the hash value matches the number of zeroes specified by this variable. If so, the mined block will be added to the end of the blockchain, however if not the nonce value of the block will be incremented by 1 and the hash will be generated again.

The purpose of this requirement is to slow down the validation process for new blocks and require some form of computational input in order to add to the blockchain. This is important for the security of blockchain, as it will require significant computational input for an attacking node or bad actor within the network to insert malicious data by attempting to modify a block. Every block up until the one they seek to modify will require rehashing and the attacker's blockchain will need to catch up to the longest valid blockchain as well if it wants to become the longest 'valid' blockchain.

The number of zeroes can be altered depending on the implementation and increasing it will exponentially increase the time it takes for a block to be generated. For this reason this variable can be seen as a way of determining the difficulty of hashing a valid block. In blockchains like Bitcoin it is standard to adjust the mining difficulty to keep it relatively constant by using a moving average, however in this implementation it has been left static.

Supporting Functions

In this implementation, the blockchain object makes use of the following supporting functions to complete the required functionality.

- is_valid(self): One of the most important blockchain features, this function works to identify if the actual chain is valid or not. Validity is determined by checking the following three criteria. The first check is if the block is arranged in order and this is done by comparing the index of each block with the previous block. Another way of doing this would be by comparing the timestamps, however since each block is generated very fast it's a lot harder to visualize the difference in sequence according to timestamp. With the index it is much simpler since the main goal of this check is just to determine that everything is outlined in the correct order.

  The second check is done to compare the "prev_hash" value contained in a block with the actual hash value of the previous block. The "prev_hash" value is designed to contain the hash value of the previous block in the chain so that each block has a record of the block that came directly before it. If the prev_hash value and the hash value of the previous block are not identical then it means someone attempted to change the data however they weren't able to rehash the entire chain. The one exception of this rule is the genesis block, since it is the first block created in the chain it is initialized with a fairly arbitrary previous hash value as there is no actual previous block within the chain, and therefore we don't try to compare it to one.

  The third check is to see if each block within the chain is actually valid. In order for the chain to be valid, every block within the chain also needs to be valid. This step is completed by running the "is_valid()" helper function within the block object and more details can be found in that section, however the main takeaway is that to check this parameter we are looking to make sure that by regenerating the hash value for this block we're not changing the hash value at all. If the data within the block header has been manipulated at all the hash value will

change and this function will return false for this check. With a weaker hashing algorithm, it would be possible for the hashes to match up, allowing an attacker to change data without triggering an invalid response, which is why it is so important to use as strong as a hashing algorithm as feasible.

This implementation walks through each check, prints the values that are being compared, and shows which check specifically fails if the chain is invalid. If the chain is valid it returns true and prints a message that the chain has been validated.

- get_last_block(self): This is a very simple function which just returns the last block within the blockchain. This has been included because the last block within the chain is very important with respect to increasing the size of the chain and it was easier than having to rewrite the specific code every time I needed to access it.
- print_blockchain_basic(self): Another simple function, this is used primarily for debugging to print a simplified list of blockchain information. This became easier for me to use to visualize the status of the blockchain during testing and this also is used heavily during the demonstration portion of the project.
- find_by_index(self,search_index): Another simple function, this only returns the block at the specified index or nothing if that block doesn't exist. This is useful for accessing specific blocks within the chain and is used as part of the demo to manipulate data which is supposed to be secure.
- add_block(self,block): The purpose of this function is to append any new blocks to the end of the existing blockchain once they are finished being mined.

While not explicitly functions of the blockchain object, there are several functions in the *mining.py* file that are useful for the blockchain object that have been kept separate in order to improve readability.

- mine(chain): This function takes the chain as input and mines a block which is added on to the end of the chain. For this new block the index is incremented by one from the previous last block, a new timestamp is generated, and the previous hash is taken from the previous last block. The nonce is initialized to 0 and the data is an arbitrary placeholder, "This is block #X", where X is the index of the new block. By creating a new block with this information, a hash value is generated and a new hash value continues to be generated until the proof-of-work concept is satisfied. This means that the hash value is checked for a specific number of leading zeroes, and this specific number of leading zeroes is set by the blockchain object for this specific implementation. If the hash value does not start with this number of leading zeroes, the nonce is incremented by one and a new hash value is generated for the block. This continues until the number of leading zeroes in the hash value matches the specific number of leading zeroes expected by the blockchain. This is referred to as the proof-of-work concept, where the blockchain is deliberately slowing down the process of generating a new block so that a certain level of processing effort is required to create a new block. This is an important part of the security of a blockchain, as in order to manipulate the data within a blockchain an attacker will have to replicate the entire processing

effort of the blockchain up until that point while simultaneously catching up with the current position of the existing longest valid blockchain. This function is useful for creating placeholder data and is one of the functions I set up to initialize the blockchain.

- mining_input(chain): This function is identical to the previous function, however this time it requests input from the user as the data variable in the form of a string. This data can be anything as long as it is in string format and is useful to show that a blockchain can be used to validate any form of data, not just cryptocurrency transactions. For example, input can be the form of a signature and as long as that signature is recorded in the blockchain it can not be manipulated or altered.

In addition, the *consensus.py* file contains a function which was written for a blockchain to use to find the longest valid chain on a network, and then adopt this as the best chain. For a blockchain to work the consensus protocol is important since it will be necessary for all nodes to operate from more-or-less the same blockchain with slight variations as new blocks are mined. Each node is responsible for sharing the status of it's chain with other nodes, and if there is any discrepancy a node will adopt the best chain from amongst its peers. It is also possible that nodes on opposite sides of the network have each mined a new block before they get the chance to communicate with each other. In this case it's possible for both chains to be different but still valid, so they need some way of resolving these discrepancies, potentially by merging the two chains based on mining timestamp.

- find_best_chain(peers): This function takes a list of peers for a given node and searches for the longest valid chain within the network amongst these peers. It then returns the best available chain within the network, as determined by the blockchains length and validity. This helps nodes which have previously gone offline or missed out on some of the communications catch up with the longest valid chain in terms of progress without having to redo the work of mining each block. This is important because it helps spread the longest valid chain without exposing the blockchain to any security vulnerabilities.

The *initialize.py* file is used to store the functions for both creating the genesis block of the blockchain as well as the other steps necessary to create a new blockchain. These functions are only called once per iteration, after a blockchain has been created there is no need to create a new one unless the existing blockchain data has been fully erased. In a deployed system, this would require all of the nodes to each lose their copy of the blockchain or have it overwritten so a new blockchain would need to be created, showing the strong fault tolerance attributes of a blockchain ledger.

- create_genesis_block(chain): This section of code is called by the following function to create an instance of the first block within a chain. Since it is the first block, this is treated as a special case. Most notably is the fact there is no value to store in the prev_hash variable, as this being the first block means that there is no hash value available for a previous block. For this reason the value is initialized to a generic value and must be treated as a special case by any validation functions.

Other than this, the genesis block follows all the same rules as a normal block within the chain, meaning it stores data (some placeholder value in most circumstances), a timestamp, an index (0), a nonce, and a generated hash value to comply with the proof of work concept.

- initialize_blockchain(node_id): When a node is created, if it does not have any peers it will initialize a new blockchain that it will work off of until it connects with peers and receives a better version of the blockchain from them. By initializing a blockchain, it creates the appropriate local directory to store the block data in and calls the previous function for creating a genesis block. Specifically for the purposes of demonstration. this function has also been set up to mine two more blocks containing generic information. The reason for this is it is much easier to demonstrate blockchain functions when the blockchain has multiple blocks, however this is not a necessary step.

C. Block

Contents

The block object is a way for the blockchain to store discrete information. Each block represents a distinct record within the blockchain, and once a block has been created and added to the blockchain it cannot be changed or destroyed. The main information that will be recorded by each block is as follows:

- index: This refers to the block's position within the chain. While not explicitly necessary, the index is useful for tracking the order of blocks.
- timestamp: This is the time the block was created. Like the index, this is useful for ordering the blocks within the chain and is also a key component of the hash value.
- prev_hash: Each block stores the hash value of the previous block as a way of maintaining the validity of the chain. If the hash value of the previous block does not match the "previous hash" value of the current block, it means that there has been an attempt to alter data within the block chain.
- data (i.e.., transactions): This is the actual data or transactional message that is being stored in the block. The most popular use in deployed systems is to store transactions related to cryptocurrency, however this could be anything that the user wants to have a signature of or prevent others from altering in the future. For example, a patient or hospital may want to maintain a secure record of a doctor's diagnosis or prescription history which others are incapable of manipulating.
- nonce: The nonce is used as part of the hashing "puzzle" in Proof-of-Work based systems. In this implementation, we require each block to begin with a specific number of zero bits in their hash value. In order to slow down the creation of more blocks, the nonce is incremented with every attempt to solve the puzzle as a way of creating a variation in the hash value.
- hash: The actual hash value of the block, this is created by hashing together the contents of the block. The previously discussed elements of index, previous hash, timestamp, and nonce all combine to form the header for the block. Using a

hashing algorithm, in this case SHA-256, the block header is hashed and this value is assigned to the block.

Supporting Functions

In this implementation, the block data structure makes use of the following supporting functions to complete the required functionality for the blockchain to work as expected.

- block_header(self): This function returns the contents of the block which are going to be used to determine the hash value as a concatenated string. The specific components are the block's index, the hash of the previous block, the block's data, and the timestamp of the creation of the block. This is referred to as the block's header.
- hash_generator(self): In this function, hash value is generated for the block based on the block header that has been created. The hashing algorithm used for this implementation is SHA-256, however any hash value may be used (even ones that are not very secure) to create a blockchain. The security of the hashing algorithm is a large contributing factor towards the security of the blockchain overall.
- are_blocks_equal(self, other_block): This function does an element-wise comparison between the contents of two blocks to determine if they are identical.
- is_valid(self, chain): This function regenerates the hash value of the block based on the data within the block, and checks to see if the hash value still satisfies the proof-of-work concept (ie., are there still the expected number of leading zeroes). If there are, it returns a valid block, however if there are not the block is invalid. The way this is currently written it is almost impossible, however if the updated hash value still contains the correct number of leading zeroes then there is a chance an invalid block is passed back as valid. Even if this does happen other validity checks will still catch this and cause the chain to be invalid, however I wanted to call attention to this specifically. Another way of writing it would have been to compare the updated, regenerated hash to the original hash within the block and see if this has changed however at the time I originally wrote certain functions I wasn't sure how I wanted the ledger to handle being fed bad data. The other validity checks will be talked about more in the following section.

IV. Demo Functions

To assist in demonstrating the functionality of this implementation I created a script labeled *demo.py*. This script contains several supporting functions which each demonstrate useful information related to the blockchain. When this script is executed, it starts by creating two nodes and assigning each other as peers. When the first node is created, the blockchain is initialized by creating a special genesis block and then by mining two other blocks containing generic information. This code has been included in the *initialize.py* file and can be altered as necessary. The genesis block is unique as it is the first block in the blockchain and has been assigned an arbitrary value for the previous

hash variable. The data it contains has also been restricted to “First block”. Immediately after the genesis block the two blocks are mined to give the blockchain a bit more depth, as well as provide more blocks which can be manipulated for the purpose of demonstration. The data contained within these blocks is by default a label describing the index of the block within the chain. After this step, the contents of the chain are printed and a prompt is displayed waiting for input from the user to determine which set of functions will be demonstrated next.

- validity_demo(node): Called via ‘validity’, this function loads the chain from the local files if it is not already available and then runs the *is_valid()* function from the *blockchain.py* file. The object of this demonstration is to showcase the checks which are performed to determine if a chain is valid or not, and the output print statements will direct the user to where any invalid information is detected.
- mining_demo(): Called via ‘mine’, this function loads the chain from the local files if it is not already available and prints the data within the blockchain to showcase its original state. The function then waits for the user to input customized data that they would like to add to the blockchain and feeds this as input to the *mining_input()* function from the *mining.py* file. After the new block has been mined, it is added to the chain, saved to the local files, and the chain is re-synced. The function will then print out the updated data within the blockchain.
- corrupt_demo(node): Called via ‘theft’, this function loads the chain from the local files if it is not already available and then prints the original state of the blockchain data. It then asks the user for the index of the block they would like to alter the data of, and afterwards what new data they would like to add into the selected block. Once this is done, it will print the updated state of the blockchain. At this time the hash value for the block has not been updated, however by running the validity check following this function it can be shown that the validity function will be stopped at this block and for what reason. Prior to running this function, it is advised to create a backup version of the blockchain that can be rolled back to, I added this functionality however did not make it automatically take place for the purposes of demonstration and testing. This can be done via the ‘store’ and ‘load’ commands.
- print_chain(node): Called via ‘print’, this function loads the chain from the local files if it is not already available and then prints the blockchain data using the *print_blockchain_basic()* function from the *blockchain.py* file.
- consensus(node): For the given node, this function accesses the peers of that node and finds the longest valid chain within this list of peers using the *find_best_chain()* function. Once the best chain is returned, it checks to see if it is better than the entered nodes current chain, if it is this function replaces the current version of the chain with this better version within the node. The node then broadcasts the best version of the blockchain to all of its peers so that they can update their stored versions as well. This is a limited way of resolving disputes across chains, as it is possible for two chains to be different however have equal lengths and this function will not have them reach a consensus, however since blockchains are constantly adding new blocks one should overtake the other eventually.

When the demonstration is over or if alterations need to be made to the supporting Python files the script can be ended by calling 'end'. Also featured is a set of functions which are used to create a backup version of the local *chaindata* directory. The *chaindata* directory is where the local blockchain files are stored for a specific node and where the *sync()* function loads the blockchain from. By creating a backup version by calling 'store', the user can rollback the state of the blockchain by calling 'load'. This functionality was originally added to restore the blockchain to a valid state when data was purposefully manipulated, however I made this something that should be called manually to provide more control over testing and demonstration.

V. Conclusion

Overall if nothing else this was an informative project for learning more about blockchain and how it can be implemented. As cryptocurrency becomes increasingly popular the use of blockchain technology will continue to grow so its important to build a deeper understanding of how it works. Just by reading it is difficult to understand some of the concepts blockchain is founded on and I found it was a lot easier to follow by actually writing the code, researching supporting reference material, and thinking about this from the perspective of how I would demonstrate a blockchain to someone who has no working background knowledge at all. One of the biggest challenges was that I consider myself a novice programmer when it comes to Python as most of my experience is in C, C++, and Java. I decided to proceed with Python because it provided a valuable learning opportunity as well as because it was the language I was able to find the most supporting reference material for of which I had a passing knowledge. If I had more time and the opportunity to continue working on this project the first thing I would want to do is flesh out the mining concept more. Currently this implementation is limited to specific input commands from the user, however it would be optimal for a realistic system to have transactions occur automatically, be mined by multiple nodes simultaneously, and have new blocks then automatically be broadcast to all nodes within the network. For the purposes of demonstrating a system, however, this would make it harder to follow the process and for this reason it was not implemented. Another improvement on the design would be to allow users to create nodes on their own personal computers and have them send/receive information across the internet. This is what makes viable blockchains such as Bitcoin so important, however because this does not expand on functionality specific to blockchain and would be a significant investment of time and effort this was not included here.

VI. References

[1] Nakamoto, S. (n.d.). *Bitcoin: A peer-to-peer electronic cash system*. bitcoin.org. https://bitcoin.org/bitcoin.pdf