Lecture 5: Bitcoin System-Putting it All Together

Principles of Blockchains, University of Illinois,
Professor: Pramod Viswanath
Scribe: Suryanarayana Sankagiri, Xuechao Wang

February 9, 2021

Abstract

In this lecture, we cover some aspects regarding transactions in Bitcoin. Transactions give semantic meaning to the decentralized ledger enabled by a blockchain. We discuss how Bitcoin gets its monetary value and how money is transferred from one user to another. We highlight the different safety checks that users can perform to prevent any form of cheating. This leads to the notion of validating a block. We argue that validating blocks is a storage and computation intensive process. and show how Bitcoin can support light clients that perform a limited extent of validation. Finally, we take a high-level view and discuss how a blockchain can act as a distributed computer, running arbitrary programs termed as smart contracts.

In any currency/banking system, there are some basic requirements that the system must provide for. We list them here:

1. There should be a unit of currency/money.
2. There should be a standard way of keeping accounts, i.e., keeping track of how much money each person owns, and transferring money between accounts.
3. No user should be able to create new money from thin air. Put differently, there should be a fixed amount of money in the system at any given time, and new money should be introduced in a systematic manner.
4. A user should not be able to spend more money than he/she owns. There should be a way to verify whether or not this happens.
5. One user should not be able to spend someone else's money (at least, not without their permission).

Let us see how the Bitcoin system provides these features.

Bitcoin and Satoshi

The basic unit of currency in the Bitcoin system is, simply, Bitcoin. The smallest denomination of a Bitcoin is called a Satoshi. It is equal to 10^{-8} Bitcoins. All transactions must be some integer multiple of a Satoshi. Just as all other currencies in the world have exchange rates, there is an exchange-rate between Bitcoin and the dollar. As of today, February 9 2021, one Bitcoin is worth $48,000$ US dollars. Due to various factors (including greed and speculation), the exchange rate between Bitcoin and dollars is very volatile. Whether Bitcoin should be thought of as a currency (like the US Dollar) or a store of value (like the precious metal, gold) has been debated widely; the mainstream view is that Bitcoin is a combination of both. Economically valuing Bitcoin, both in the

short and long terms, is an active area of research. In this lecture we will see aspects of the Bitcoin system that helps understand the economic aspects of Bitcoin, both as a currency and a store of value.

Transactions

In ordinary parlance, the term **transaction** refers to an exchange of something of value. In the context of Bitcoin and cryptocurrencies, a transaction is simply a message that specifies the transfer of money from one entity to another. In fact, transactions are the data-values that get recorded on the blockchain. The blockchain as a ledger is therefore an ordered list of transactions. From this publicly verifiable ledger, any user can detect whether transactions are made according to certain rules or not, thereby lending credibility to the ledger and the currency.

In Bitcoins, transactions have a well-defined structure, which we elaborate upon below. One can take a look at the structure of real Bitcoin transactions here.

Addresses Naively, in a currency system, there should be some notion of an account; a transaction notes the transfer of money from one account to another. In Bitcoin, the notion of accounts is replaced with that of *addresses*. Bitcoins are allocated to addresses, and these addresses are also used to decide where Bitcoins will be sent to, in a transaction. What exactly is an address and how is one generated? In Bitcoin, an address is simply the hash of a public key. Recall the notion of digital signatures, and that of public and private keys. New pairs of keys, and thus new addresses can be generated at will by a single user.

One idiosyncrasy of Bitcoin is the following. In order to receive coins, a user needs to only publish its address, not its public key. However, to spend coins, a user must also reveal its public key. This is explained below.

Transaction inputs and outputs A transaction in a Bitcoin is a statement that records a transfer of money from one address to another. More broadly, a transaction records a transfer of money from one set of addresses to another. Every transaction has a set of *transaction inputs* and *transaction outputs*. Each transaction input states the amount of Bitcoin being spent by a particular address. Each transaction output states the amount of Bitcoin being received by a particular address. In a transaction, the total money being spent should add up to the total money being received.

Signatures on transactions Each transaction must be signed by all the users that are spending money. This is a basic safety feature that prevents others from spending one's money without authorization. As mentioned above, each address has a one-to-one correspondence with a public key, which in turn has a one-to-one correspondence with a private key. A user that wishes to spend Bitcoins associated with a particular address creates a transaction appropriately, and then signs the transaction using the corresponding private key. It then broadcasts the transaction along with the corresponding public key. Anyone who that sees a signed transaction can verify whether it was signed by the person owning the address (and thereby, the coins in the address). Thus, signatures help other users validate transactions. More generally, if a transaction spends Bitcoins from multiple addresses, there must be signatures corresponding to each of these addresses.

UTXOs So far, we have seen how transactions are used to transfer money from one address to another. What prevents a user (i.e., an address) from spending more money than it has? One method would be to keeping track of the balance of each address, adding or deducting its value as money is received/spent from the address. Bitcoin adopts a different approach; here, every transaction input must be a transaction output of an earlier transaction. Linking transaction inputs to past inputs provides a proof that the address indeed has sufficient money to spend.

When a new transaction output is created, we say that it is *unspent*. At some time in the future, it gets consumed as part of transaction input, at which point it is *spent*. A valid transaction must only include *unspent transaction outputs* (UTXOs) as its inputs. While honest users will always ensure this, a dishonest user can try to *double-spend* its money. In order to prevent this, honest users must keep track of the set of UTXOs at all times, and must validate every new transaction in the blockchain against this set. We elaborate more on this in later sections. The above method of validating transactions is called the **UTXO model**. A more natural technique would be the **account-based model**, where the balance is maintained for each address. This latter model is adopted in other cryptocurrencies such as Ethereum (and also in regular banks).

We now see why a Bitcoin transaction allows for multiple transaction inputs and outputs. First, let us consider the need for multiple outputs. Suppose a particular user owns a single Bitcoin address, to which it has received 2 Bitcoins in a particular transaction. Even if it wants to spend a fraction of that money (say, 1 Bitcoin), it must spend the only UTXO it has, which is of 2 Bitcoins. In such a case, it creates two transaction outputs, one to the address it actually wants to send the money to, and the other to itself (the change). The latter output could be to the original address, or to a new address. Next, let us consider the case of multiple inputs. Suppose an address (i.e., user) has received 1 Bitcoin each in two different transactions, and it would now like to pay 2 Bitcoins to another address. It can then include two transaction inputs in a single transaction in order to pay this amount.

Cryptocurrency wallets In reality, keeping track of UTXOs, and measuring them up for each transaction is difficult. In addition, a single user ought to spawn new addresses regularly, for the sake of maintaining anonymity. These addresses (and the corresponding keys) must be generated carefully, without revealing even a hint of the private key. Further, they must be stored securely. All these functionalities are taken care of by a *cryptocurrency wallet*. Wallets are simply software that perform a lot of these tasks in the back-end, allowing users to transact in Bitcoin as one would using a bank account. Using a wallet requires trusting the software of the wallet. In principle, one can participate in the cryptocurrency system without the use of a wallet, but most users use a wallet.

Transaction fees We mentioned above that the total value in a transaction's inputs must add up to the total value in its outputs. In reality, the sum of values in the output is slightly lower than the inputs. The remaining amount, called the **transaction fees**, is claimed by the miner of the block that includes this transaction. The transaction fees are an incentive for a miner to include a particular transaction in the block being mined. Transactions with higher fees get included faster in the blockchain, while those with lower fees get added later. The fees vary with time, and are often calculated automatically by wallets according to a particular fee rate, measured in Satoshi per kilobyte. You can learn more about transaction fees and fee rates here. Roughly speaking, fees are of the order of ten dollars per transaction.

Coinbase transactions The preceding discussion is on how money is exchanged by users in the Bitcoin system. How is money introduced in the system in the first place? The answer is simple: new Bitcoins are generated with every new block. Every block includes a special transaction, called the coin-base transaction, in which the miner gets for itself a fixed number of Bitcoins. Initially, the rewards were 50 BTC per block. Every $210,000$ blocks mined, or about every four years, the reward given to Bitcoin miners for processing transactions is cut in half. So far, there have been three halvings, and the current reward is 6.25 BTC per block. Block rewards will continue till the year 2140, after which there will be no new Bitcoin introduced in the system. The total volume of currency that will be ever be used is capped at 21 million, of which around 18.5 million coins are already in circulation. Coinbase transactions, along with transaction fees, are an additional incentive mechanism for Bitcoin users to actively participate by mining. In the initial years of Bitcoin, coinbase

transactions formed the major component of the rewards; with time, the contribution of transaction fees is catching up.

Figure 1 shows how the Bitcoin block reward decreases by half every four years. We are currently at 6.25 Bitcoins per block. The figure also shows the total number of Bitcoins in circulations. The halving scheme ensures that the total number of Bitcoins ever produced will taper off to a total of 21 million Bitcoins.

Figure 2 shows the same metric as in Figure 1, but the reward is now measured in dollars instead of Bitcoin. This figure is useful to understand the incentives for mining. The block rewards (along with the transaction rewards) must offset the cost of mining. As more users join the system with time, a particular miner must compute more hashes (and thus must spend more on electricity) to mine a block. Block rewards (in terms of dollars) have gone up too, providing an incentive for users to mine in spite of the increasing difficulty. There is a subtle balance between the various economic factors that govern the level of mining power in the Bitcoin system.

Transaction mempool Whenever one user wants to pay another using Bitcoin, it generates an appropriate transaction, signs it, and broadcasts it through the entire network. Ultimately, the transaction is simply a message (a string of bits) in a particular semantic form. The size of a message is a few kilobytes; the exact size depends on the number of transaction inputs and outputs. The transaction is propagated across the Bitcoin network using the diffusion protocol. Miners keep looking out for new transactions and add them to their memory, called the **mempool** (they do so after verifying the signature on the transaction). At any given point in time, a miner is working on a particular block, which contains some transactions from its mempool. A miner can include new transactions from its mempool into its working block at will (or even remove them). As such, it would like to include as many transactions as possible in order to maximize its sum total transaction rewards. However, there is an upper bound on the total size of a block. Thus, a miner prioritizes blocks with a higher transaction fee rate, i.e., a higher fee with a smaller size. A transaction with a high enough transaction fee is included immediately by all miners into their working block. The lucky miner who finds the proof-of-work first gets to "claim" the reward; again the claim on the reward is only honored when the block is confirmed, i.e., buried deep enough in the longest chain.

Note that there is a certain latency between the transaction first being issued and the transaction being confirmed on the blockchain. The first factor behind the latency is that it takes time for a newly issued transaction to be included in any block; this latency can be reduced by increasing the mining fee. The second factor is that it takes time for a block that includes the transaction to be buried deep enough. Here, a user may trade-off latency with security (in Bitcoin). Other blockchain designs, which we explore in future lectures, may not have this trade-off.

Validating a block

The state of the system In the above discussion, we covered many important points explaining how Bitcoin enables a money-exchange system. One issue that was skirted is, how do users verify that a single coin is not spent twice? In the context of Bitcoin, this issue boils down to verifying that a transaction's inputs have not been spent already. As such, the only way to verify this is to keep track of the set of UTXOs (unspent transaction outputs) at all times. This set is often referred to as the **state** of the system. Note that these outputs are for those transactions that are already in the blockchain, and have not been spent in the blockchain. Thus, the state changes every time the longest-chain grows (or more generally, changes). It is important to note that the state is separate from the mempool, but is linked in the following way. Honest users build a block from transactions in the mempool. However, they only choose those transactions that are valid, and they check the validity using the information in the state. In an account-based model, the state would simply be the roster of all account balances. More generally, the state of a blockchain system contains a summary

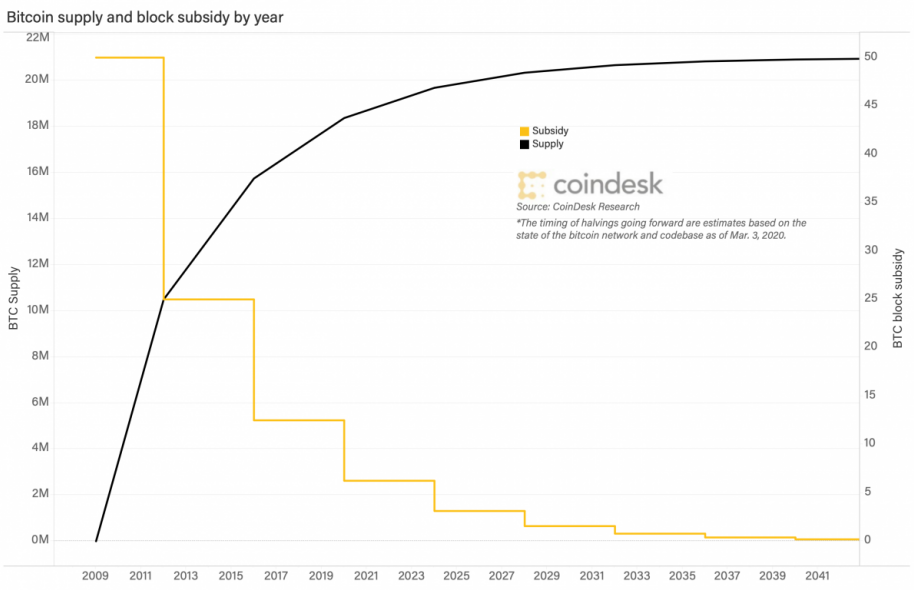

Figure 1: Bitcoin block reward in bitcoins. Figure sourced from here

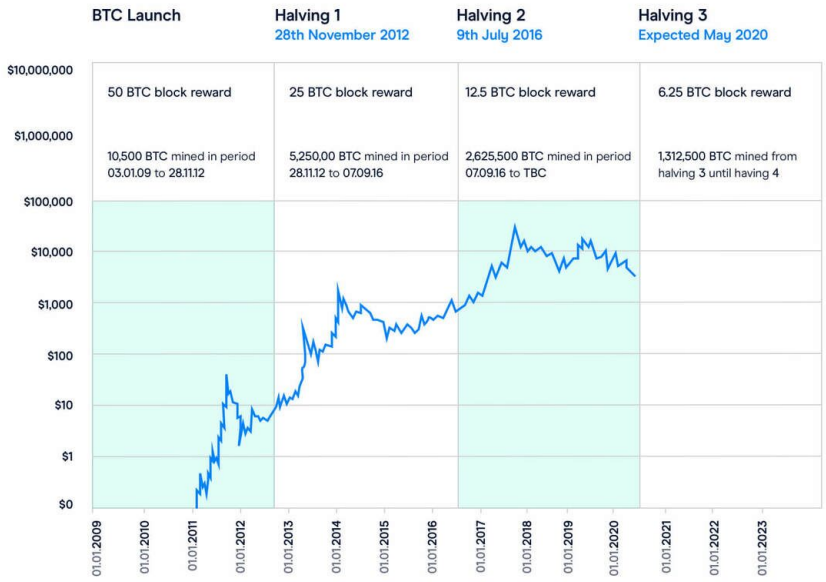

Figure 2: Bitcoin block reward in dollars. Figure sourced from here

of all the entries in the ledger thus far, which can be used to validate new entries.

Figure 3 shows the number of UTXOs in the Bitcoin system as a function of time. Clearly, it has grown many-fold since the beginning, and it will continue to grow as Bitcoin becomes more popular. With each UTXO being a few hundred bytes, the size of the UTXO set is now over 5 GB. As such, this might seem like a moderate amount which can be easily stored. However, each node must perform many read, write and delete operations on this set. Thus, the set must be stored in the memory (preferably, on-chip memory) rather than on the disk. Given this requirement, the UTXO size is too large for regular devices to store; it requires specialized hardware. The large UTXO size is another dimension of the scalability challenge.

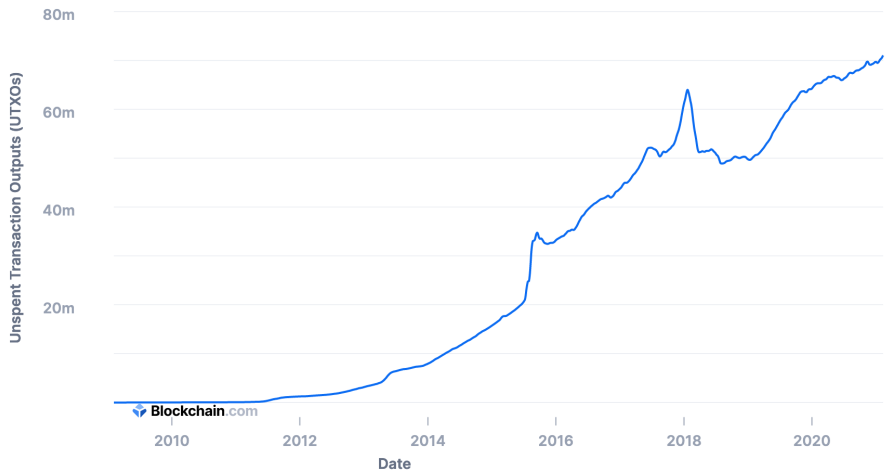

Figure 3: Number of UTXOs over time in Bitcoin. Figure sourced from here

Validating blocks When a miner receives a new block, it must perform certain sanity checks, of which two are important. First, it checks whether the proof-of-work meets the required threshold. Second, it checks whether the transactions in the block are consistent with the state. This means that it must check whether every input in every transaction belongs to the state or not. If it does, the miner accepts the block, removes the relevant transactions from its mempool, and also updates the state. Updating the state involves deleting spent transaction outputs and adding the newly generated ones. While constructing a new block to mine on, the same validation steps are taken preemptively.

Light nodes and stateless clients Storing the state of the system and validating each transaction in every block requires storage and computation power. Bitcoin allows for light nodes/clients, who can participate in the system with much lesser computation and storage power. A light client merely verifies the proof of work on blocks. It may further selectively verify the validity of transactions, especially those that concern itself. As such, a light client does not store the state of the system which prevents it from validating transactions.

There are proposals to augment the functionality of light nodes, by creating so-called **stateless nodes**. These nodes can validate transactions/blocks without storing the full state at all times. They merely download the requisite state information to validate a block, one block at a time, and then delete it once the block is validated. In order to do so securely, the state is stored in the form of an accumulator, which we discussed in Lecture 2. The stateless client option is being pursued

vigorously by Ethereum. See, e.g., the discussion here. A fully functional stateless client for Bitcoin, which makes use of Merkle tree accumulators, has been built by the UTREEXO project.

Smart contracts

So far, we have seen how blockchains can be used to implement a currency system. Essentially, the blockchain as a ledger records payments from one party to another, thereby keeping tab of all parties' balances. Transactions are simply messages recording the transfer of money. However, blockchains are much more versatile; they can also be used to run *smart contracts*. Smart contracts are programs that are run by the peers in the blockchain system. The notion of a transaction is broadened to include these pieces of code as well. Smart contracts come into play when two parties want to exchange money subject to some terms and conditions. In the physical world, they would draw up a contract. A trusted third party/authority would be needed to ensure that both parties follow the contract. When run on a blockchain, smart contracts eliminate the need for a single trusted third-party. Instead, the decentralized trust of the blockchain (in other words, the honest majority of the system) ensures that the contract gets executed correctly.

In this lecture, we only focus on smart contracts in Bitcoin, which are rather limited in scope. Ethereum, a subsequent cryptocurrency, allows for much more diverse programs as smart contracts; we will learn about that in a later lecture.

Scripts in Bitcoin Smart contracts are simply called scripts in Bitcoin. To understand scripts, we need to broaden our understanding of transactions from merely statements recording the transfer of money. As we saw before, a transaction consists of inputs and outputs, with outputs recording the transfer of Bitcoins to some address. What we did not mention before is that every transaction output includes a script, consisting of *opcodes*, which specify conditions that must be satisfied in order to spend the coins mentioned in the output. The default condition is that the spender of the output must provide a public key that hashes to the pertinent address, and must sign the message with the corresponding private key. Earlier, we presented this as a sanity check; however, this is explicitly specified as a script in every regular transaction.

Bitcoin scripts also allow for additional conditions, such as requiring that the transaction is not spent until a particular time, requiring that a transaction is spent by a particular time, requiring a message that is signed by multiple private keys, etc. An interesting and useful script that emerges from a combination of these conditions is that of a hashed timelock contract (HTLC). HTLCs are useful in setting up escrow funds. If Alice wants to pay Bob in exchange for some other good, then Alice would like to pay Bob only after she is guaranteed to get the good from Bob, and Bob would like to release the good only after he is guaranteed to be paid by Alice. An HTLC enables such terms to encoded as a smart contract on a blockchain. You can learn more about HTLCs here. A full list of opcodes used in the Bitcoin script is given here.

Midterm Project: Rust Implementation of Bitcoin Clients

In this project, you are going to build a simplified Bitcoin client with full node functionality. The goal of the client is not to run in Bitcoin mainnet or any public testnet. Instead, the goal is to run it inside your team and let you have fun with it. You have plenty of freedom of designing and implementing this project. We provide some code in the Gitlab repository. The project will be split into 6 weeks with checkpoints every week as listed below.

1 Week 1

In this week, you are going to finish the **Block** struct and the **Blockchain** struct.

1.1 Block

You need to define a **Block** similar to that in Bitcoin. We require that a block must include:

1. parent - a hash pointer to parent block.
2. nonce - a random integer that will be used in proof-of-work mining.
3. difficulty - the mining difficulty, i.e., the threshold in proof-of-work check.
4. timestamp - the timestamp when this block is generated. This is used to decide the delay of a block in future part of this project.
5. merkle_root - the Merkle root of data (explained below in 6.).

The above fields are also known as **Header**. We suggest (but not require) you to create a struct **Header** to include them.

6. data/content - the actual transactions carried by this block.

We suggest (but not require) you to create a struct **Content** to include the content.

1.2 Blockchain

You need to finish a struct named **Blockchain**, which contains the necessary information of a direct acyclic graph (DAG) and provides functions related to the longest chain rule. The following functions are required:

1. new() - create a new blockchain that only contains the information of the genesis block. (Define genesis block by yourself.)
2. insert() - insert a block into the blockchain.
3. tip() - return the last block's hash in the longest chain.
4. all_blocks_in_longest_chain() - return all blocks' hashes, from the genesis to the tip.

2 Week 2

In this part of midterm project, you are going to implement the **mining** module of Bitcoin client. The mining module, or miner, will produce blocks that solve proof-of-work puzzle. The programming goal for this part is to prepare the miner and implement the main mining loop.

2.1 Preparation for miner

You need to add required components to **Context** struct in *src/miner.rs*. Specifically, the miner calls *blockchain.tip()* and set it as the parent of the block being mined. After a block is generated, it needs to insert the block into blockchain. Hence, in this part, you need to add blockchain into miner **Context** struct. It is running in another thread, hence we need the thread safe wrapper of blockchain.

2.2 Main mining loop

The main mining loop is the loop that is trying random nonces to solve the proof-of-work puzzle. To build a block, you need to gather a block's fields. In a block header, the fields are gathered as follows,

1. parent - use *blockchain.tip()*
2. timestamp - we suggest to use millisecond as the unit rather than second, since when we measure block delay in the future, second may be too coarse.
3. difficulty - it should be computed from parent and ancestor blocks with some adaptive rule. In this project, we use the simple rule: a static/constant difficulty. This rule just means the difficulty of this block should be the same with that of parent block. You should be able to get parent block's difficulty from blockchain.
4. merkle_root - compute it by creating a merkle tree from the content.
5. nonce - generate a random nonce in every iteration, or increment nonce (say, increment by 1) in every iteration.

As for the block content, you can put arbitrary content, since in this step we don't have memory pool yet. You can put an empty vector, or some random transactions.

After you have all these fields and build a block, just check whether the proof-of-work hash puzzle is satisfied by

```
block.hash() <= difficulty.
```

If it is satisfied, the block is successfully generated. Congratulations! Just insert it into blockchain, and keep on mining for another block.

3 Week 3

In this part of midterm project, you are going to implement the **network** module of Bitcoin client. The network module is in charge of communicating with other nodes/clients. It forms the peer-to-peer (p2p) network and uses gossip protocol to exchange data, including blocks and transactions. Transactions will not be covered in this part. You will define a few message types and the behavior when they are received.

3.1 Message types

You need three message types. You can define them in *src/network/message.rs*.

1. **NewBlockHashes**: similar to *inv* in lectures, inventory message containing block hashes.
2. **GetBlocks**: similar to *getdata* in lectures, which asks for the same block as NewBlockHashes.
3. **Blocks**: similar to *block* in lectures, which sends the entire block including header and content.

3.2 Gossip protocol

You need to define the gossip protocol, i.e., the behavior when messages are received, in *src/network/worker.rs*.

First, you need to add thread safe wrapper of Blockchain into **Context** struct in *src/network/worker.rs*. Notice that the server we provide is a multi-thread one, so please be careful with thread safety. Then, you can define the gossip protocol introduced in the lecture.

1. For **NewBlockHashes**, if the hashes are not already in blockchain, you need to ask for them by sending **GetBlocks**.
2. For **GetBlocks**, if the hashes are in blockchain, you can get these blocks and send them by **Blocks** message.
3. For **Blocks**, insert the blocks into blockchain if not already in it.

4. Optional. If a block's parent is missing, put this block into a buffer and send **Getblocks** message. The buffer stores the blocks whose parent is not seen yet. When the parent is received, that block can be popped out from buffer and inserted into blockchain.

3.3 Combine with miner

We've defined **NewBlockHashes**, so when miner successfully generates a new block, just broadcast that message.

4 Week 4

In this part of the midterm project, we will be combine last 3 week's work to make a functioning data blockchain. Most of this week's work will be combining mining, network and blockchain module. You will need to add PoW validation and a block buffer to handle orphan blocks.

4.1 Relay blocks

Here we ask you to extend the gossip protocol so that the propagation of blocks can be faster. In *src/network/worker.rs*, you need to make a broadcast of a **NewBlockHashes** message when receiving new blocks in **Blocks** message. **NewBlockHashes** message should contain hashes of blocks newly received.

4.2 Checks

When processing a new block in *src/network/worker.rs*, please add the following checks.

4.2.1 PoW validity check

Add code to check the PoW validity of a block by checking if:

1. PoW check: check if `block.hash() <= difficulty`. (Note that difficulty is a misnomer here since a higher 'difficulty' here means that the block is easier to mine).
2. Difficulty in the block header is consistent with your view. We have a fixed mining difficulty for this project, thus, this would just involve checking if difficulty equals the parent block's difficulty. (This step should be done after parent check.)

4.2.2 Parent check

1. Check if the block's parent exists in your local copy of your blockchain, if the parent exists, add the block to your blockchain.
2. If this check fails, you need to add the block in an 'orphan buffer', and also send **GetBlocks** message containing this parent hash.

4.2.3 Orphan block handler

Check if the new processed block is a parent to any block in the orphan buffer, if that is the case, remove the block from orphan buffer and process the block. This step should be done iteratively. I.e., a block makes a former orphan block be processed, and the latter makes another former orphan block be processed, and so on.

5 Week 5

This part of the Midterm project will deal with transactions. Integrate the transaction structure inside the block content, add network functionality to transaction propagation and adding a transaction mempool to be used by the miner to include transaction content in the block being mined.

5.1 Transaction network messages

Add the following new messages:

1. **NewTransactionHashes**: similar to **NewBlockHashes**.
2. **GetTransactions**: similar to **GetBlocks**.
3. **Transactions**, similar to **Blocks**.

5.2 Transaction format

You are free to choose any format for transaction structure. We recommend using a transaction structure that is either compatible with the UTXO model in Bitcoin or the account based model in Ethereum.

- UTXO model transaction: input contains the hash of previous transaction and the index; output contains a recipient address and a value. It can support multiple inputs/outputs in a transaction.
- Account based model transaction: it should contain a recipient address, a value, and a account-nonce. It only supports single sender and single receiver. This should be simpler to implement than UTXO model.

Now it's time to add **Signature** to transaction. Then append the public key and the signature to transaction by

- either create a struct **SignedTransaction** that contains the transaction, the public key, and the signature,
- or define fields in transaction that store the public key and the signature.

5.3 Transaction Mempool

Create a transaction **Mempool** structure to store all the received valid transactions which have not been included in the blockchain yet. If a new transaction is received/generated, add it to the mempool. **Mempool** will also be used by the miner to include transactions in the blocks being mined. The miner will add transactions in the mempool to the block till it reaches the block size limit. You are free to choose the size limit on the blocks. On processing a new block(which is not an orphan or stale), remove corresponding transactions from the mempool.

Similar to **Blockchain**, you need the thread safe wrapper on the **Mempool**.

5.4 Transaction generator

To demonstrate transaction is working well with the client, you need to add transactions into your running client. The transactions can be a simple payment in account based model, or a transaction with just one input and one output in UTXO model. You are free to choose the sender and recipient.

6 Week 6

This is the last part of midterm project, and you are going to finish the Bitcoin client. You need to maintain a state for the ledger that the blockchain creates and add all the necessary checks related to it.

6.1 Transaction Checks

In last part, you include transactions into blocks. However, in order to prevent misbehavior such as double spending, you need to add the following checks:

6.1.1 Transaction signature check

- Check if the transaction is signed correctly by the public key(s).
- In UTXO model, check the public key(s) matches the owner(s)'s address of these inputs. (This step needs struct **State**, see below.)
- In account based model, check if the public key matches the owner's address of the withdrawing account. (This step needs struct **State**, see below.)

6.1.2 Double spending check

- In UTXO model, check if the inputs to the transactions are not spent, i.e. exist in **State** (see below). Also check the values of inputs are not less than those of outputs.
- In account based model, check if the balance is enough and the suggested account nonce is equal to one plus the account nonce. This check also needs **State** (see below).

6.2 State

Ledger state, or **State**, is a collection of all the required information to check transactions.

- In UTXO model, **State** should contain all the unspent transaction outputs. The format of an unspent transaction output may contain *(transaction hash, output index, value, recipient)*. Output index refers to the index in transactions (remember transactions are multi-output.) Recipient refers to the recipient address of that output, and is used as the owner of that unspent transaction output.
- In account based model, **State** should contain all the accounts' information. It may contain *(account address, account nonce, balance)*.

6.2.1 State update

When executing a block, i.e., executing transactions in that block, we need to update the state.

- In UTXO model, remove those *inputs*, and add *outputs* to the state.
- In account based model, change account's nonce and balance. Create new accounts if you need.

6.2.2 Initial state (ICO)

You can do initial coin offering (ICO) by inserting entries into **State** struct.

- In UTXO model, add unspent transaction outputs and specify the recipients to be the addresses you control.
- In account based model, create accounts whose addresses are under your control.

6.2.3 State per block

Since there is branching/forking in the blockchain, and the longest chain may change, you need to store one copy of **State** for each block. A copy of **State** for a block refers to the state after executing the block. We recommend using a HashMap-like storage. When you check transactions, you can get the corresponding state from it. When you update state, you do the update on a new state copy, and insert it. Another way to deal with forking is to implement a reverse state transition corresponding

to a transaction, say that the longest chain changes from A-B-C-D to A-B-E-F-G, you can perform reverse state transition on blocks D and C and a forward state transition from blocks E, F, G.

6.3 Transaction generator

Transaction generator should generate transactions that pass the checks. It can read the blockchain and the state to ensure that. On different nodes/processes, transaction generator should control different key pairs.

6.4 Transaction Mempool update

After implementing state transition, ensure that the transactions in the mempool are valid with respect to the new state, this is necessary since some transactions may classify as double-spends after the state update, you may need to remove those transactions.

6.5 Conclusion

Now that you finish the last part, you have a simplified Bitcoin client! With transaction generator simulating user's transactions, the system should run smoothly and securely.