Decentralization and Democracy in Mining Pools

B. Tech Project Report Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Bachelor of Technology

by

Varenyam Bakshi and Aryan Anshuman

(190101098, 190101013)

under the guidance of

Sukumar Nandi

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, GUWAHATI

GUWAHATI - 781039, ASSAM

Abstract

In the past few years, Cryptocurrency and Blockchain technology has taken the world by storm. Peer-to-peer transactions, supply chain management, insurance and protection from money laundering are some of the numerous use cases of Blockchain technology. All blockchain networks depend on an underlying consensus mechanism verifying each new block that is added to the network. Currently, Proof Of Work (PoW) based consensus is the most widely used consensus mechanism where miners have to solve a SHA256 mathematical puzzle in order to mine a block and receive incentives. Due to unstable income and heavy hash power requirement miners tend to join Mining Pools for assured rewards. Mining Pools are significant contributors to many blockchain networks present worldwide. But these mining pools are controlled by individuals or a group of individuals who have unchallenged control over the activities and policies of the mining pool. As the size of the mining pool grows, it becomes easier for the manager(s) of the mining pool to launch a double-spending attack. Also, the individual miners are powerless and have no say in the functioning of their pool. This much control by a central authority defeats the very principle of decentralization on which the whole Blockchain Technology is based. In this paper, we present a semi-decentralized and democratic model of Mining Pool, where the power is distributed among elected leaders who are kept in check by the common miners. At last, we take a look at PoS based Blockchain and the challenges that mining pools will face in PoS based consensus networks.

Contents

Chapter 1

Introduction

1.1 Blockchain

Blockchain is a shared, immutable ledger that facilitates the process of record- ing transactions and tracking assets in a business network. An asset can be tangible (a house, car, cash, land) or intangible (intellectual property, patents, copyrights, branding). Virtually anything of value can be tracked and traded on a blockchain network, reducing risk and cutting costs for all involved. It essentially generates a distributed ledger without the requirement for a central authority to build trust or settle transactions. It also makes no assumptions. A **Hash** is a function that meets the encrypted demands needed to solve for a blockchain computation. Hashes are of a fixed length since it makes it nearly impossible to guess the length of the hash if someone was trying to crack the blockchain. The same data will always produce the same hashed value. The previous block's hash is incorporated in the next block created via the blockchain's block data structure. The application of a hash chain ensures that data stored on one block cannot be altered without invalidating succeeding blocks. As a result, tampering with data stored on the blockchain is exceedingly difficult.

Before the advent of blockchain, almost all financial transactions required a trusted third party or a central authority to validate any transaction. But after Nakamoto proposed a Proof of Work (PoW) based consensus mechanism , decentralized peer-to-peer transactions

were made resistant to double-spending attacks. In PoW, miners have to find a nonce (a random number) which, when appended to the transactions listed in a block, generates a hash value less than a certain threshold (begins with a predefined number of zeroes). The number of zeroes determines the difficulty of the problem, which is adjusted regularly in order to maintain an average block generation interval. Although PoW has been universally acknowledged as a leading consensus procedure, numerous experts have expressed concerns in recent years about:

1) Low rate of transactions per second.

2) Unsustainable energy consumption.

3) Poor scalability.

4) Diminishing mining rewards.

In response to the aforementioned performance limitations of PoW mining, blockchain researchers have been investigating new consensus mechanisms such as proof of stake (PoS), proof of authority (PoA), proof of elapsed time (PoET), and others that do not necessitate computation-intensive mining, thereby re- ducing energy consumption.

1.2 Mining Pools

A mining pool is a joint group of cryptocurrency miners who combine their computational resources over a network to strengthen the probability of finding a block or otherwise successfully mining for cryptocurrency. Individual miners set up their resources to connect. to the pool server in accordance with certain mining methodology Miners work together to mine blocks, and after success- ful mining, The block reward is distributed among miners in accordance with because of their contributions. The partnership effectively evaluates individual miners' mining efforts and ensures them regular payments according on their hashing power. Mining pools usually charge a small transaction fee for each reward obtained for offering mining services.

1.2.1 Working

Owners of mining pools, often known as pool operators, choose which transactions to include in the block and send the block templates to miners with a higher nonce (Difficulty has been lowered). The miners' objective is to find a nonce which is smaller than the threshold supplied by the pool manager (not the actual target of the blockchain) and return the block to the pool operator. This results in the miners receiving "shares" (partial Proof-of-Work; pPoW). When a miner obtains a hash value smaller than the real target of the blockchain (full Proof-of-Work; fPoW), the pool receives the reward, which the pool operator divides among the miners based on their share contribution. The pool server is linked to a few full nodes and has access to a copy of the whole blockchain database, allowing it to vali- date transactions on behalf of miners, relieving them of the burden of having a full node.

1.2.2 Rewards and Payout scheme

There are two reward component involved in mining pools.

1) **Block Rewards:** They are new coins distributed by the network for successfully mining the block. Currently, it is 6.25 BTC, which started from 50 BTC for the first block and has halved every 210000 blocks, which is approximately 4 years (since each new block is mined in 10 minutes).

2) **Transaction Fees:** It is the difference between the input transactions (sent from a wallet to the network) and output transactions (sent from the network to a wallet) included in the block header. This is greater than or equal to 0 BTC and can go up to 10% of the block reward.

Though there are over 15 reward payout schemes, here we will be describing only the four most commonly used payout schemes,

1) **Pay Per Share (PPS)** - It offers an instant payout for each share that is solved. The payment is made based on probabilistic results rather than what occurs, the miner is rewarded the block reward based on its luck.

2) **Full Pay Per Share (FPPS)** - In PPS, the reward is shared only based on the expected value of the block reward, but that does not include the transaction fees. FPPS is similar to PPS, except it pays the expected value of both the block reward and transaction fees. It calculates the average transaction fees within a certain period of time (mostly the last 24 hours or the previous day) and distributes it to the miners.

3) **Pay Per Last N Shares (PPLNS)** - It pays miners as a percentage of the shares they contribute to the total N shares. It pays the miners according to the shares contributed during the last N shares from the block that successfully gets mined.

4) **Pay Per Share + (PPS+)** - It is a combination of both PPS and PPLNS where block rewards are settled according to the PPS rule, and the transaction fee is settled according to the PPLNS scheme distributing the mining pool's actual transaction fee.

As a result, PPS and FPPS pay miners on a regular basis based on their predicted earnings; in such instances, the mining pool must bear all of the risk, hence only big pools with huge reserves may employ these payout methods. PPLNS, on the other hand, places the full risk load on the miners and compensates only when successful blocks are mined; hence, it is favoured by mining pools. Finally, PPS+ finds a balance by putting the pool's block rewards and the miner's transaction fees at risk. It is a hybrid of PPS and PPLNS, with block rewards distributed according to the PPS rule and transaction fees settled according to the PPLNS rule.

1.3 PoS Consensus Mechanism and challenges for Mining Pools

Proof-of-Stake (PoS) mining originated in the blockchain ecosystem as a more energy-efficient alternative to PoW mining. In its most basic form, a stake refers to a participant's money or network tokens. Stakes can be invested in the blockchain consensus process. From a security standpoint, PoS leverages token ownership for Sybil attack mitigation. In contrast to a PoW miner, whose probability of proposing a block proportional to its computation capacity, a PoS miner's chance of proposing a block is proportional to its stake value. In

terms of economics, PoS shifts a miner's opportunity cost from outside the system (waste of processing power and electricity) to within the system (loss of capital and investment gain). Following are the main classes of PoS consensus mechanism.

1. **Chain-Based PoS:** Chain-based PoS is a pioneering PoS system suggested by Bitcoin researchers as an alternate block generation mechanism to PoW. The gossipingstyle data forwarding, block validation rule, longestchain rule, and probabilistic finality are all kept well within the context of the Nakamoto consensus. Unlike PoW, PoS does not rely on inefficient hashing to produce blocks. A minter can only solve the hashing challenge once each clock ticks. Because the complexity of the hashing problem lowers as the minter's stake value increases, the estimated number of hashing tries for a minter to come up with a solution can be significantly decreased if her stake value is immense. As a result, PoS minimizes the brute-force hashing contest that would occur if PoW were utilized, resulting in a significant reduction in energy consumption. Examples include Peercoin and Nxt.

2. **Committee based PoS:** To create blocks, chain-based PoS still depends on hashing power. As an alternate technique, committee-based PoS follows a more regulated regime by forming a committee of stakeholders based on their holdings and enabling the committee to produce blocks in turns. In a distributed network, a secure multiparty computation (MPC) approach is frequently employed to derive such a committee. MPC is a form of distributed computing in which numerous participants, starting with individual inputs, must produce the very same outcome. The MPC process in committee based PoS implements a function which takes in the current blockchain state (individual stake values etc.) and generates a pseudo-random sequence of stakeholders (the leader sequence) that will subsequently populate the block-proposing committee. This leader sequence should be consistent for all stakeholders, with those with bigger stake values having greater probability of taking up more positions in the sequence. These leaders then participate in block proposal mechanism in the order listed in the

leader sequence. Blockchains using Committee-based PoS are Ouroboros, Ouroboro Praos, Snow White and Bentov's CoA.

3. **BFT-based PoS:** Chain-based PoS and committeebased PoS both adhere to the Nakamoto consensus paradigm in that the longest-chain criterion is still employed to ensure probabilistic block finality. In contrast, BFT-based PoS (or hybrid PoS-BFT) includes an additional layer of BFT consensus that allows for quick and predictable block finalisation. Any PoS mechanism (round-robin, committee-based, etc.) can propose new blocks as long as it introduces a consistent flow of new blocks into the BFT consensus layer. Aside from the usual approach, a checkpointing mechanism may be utilized to ensure the blockchain's validity. This eliminates the need to follow the longest-chain rule and substitutes it with the most-recent-stable-checkpoint approach. Examples are Tendermint, Algorand and Casper FFG.

4. **Delegate PoS:** DPoS is a democratic variant of committee-based PoS in that the consensus group is chosen through public stake delegation. DPoS was designed to keep the scope of the consensus group under control so that the communications overhead of the consensus protocol remained minimal. Delegates are members of consensus groups elected through a procedure known as delegation. A token holder entrusts the delegate with its own stake by voting for a delegate via a blockchain transaction. As a result, the delegate captures her constituents' stake voting power and works as their representative in the consensus process. The incentive mechanism of DPoS is meant to incentivize fair delegation and consensus involvement to ensure transaction validation and consensus security. Every delegate receives a daily vote reward according to the number of votes she holds. They also earn block rewards for the validation work they perform after reaching the consensus group. Blockchains employing Delegate PoS are BitShares 2.0, Lisk and Cosmos.

Traditional mining pools are useless for PoS consensus based blockchain network. As

hash power is replaced with stake value, architecture and working of mining pool also need to be changed. Many Chain-Based PoS networks like PeerCoin use the principle of coinage. Coinage of a cryptocurrency or token holder is the product of their total stake and the time for which they are holding the token. The more the coinage of the stakeholder, the easier the hash puzzle gets for them, thus increasing the probability of them successfully proposing a block. For a blockchain network which is using coinage, our decentralized committee based mining pool could be modified to work efficiently. The common nodes voting for a particular leader, will be transferring their whole stake to the leader. Then the leader with maximum coinage would participate in block proposal process. Although the above approach might seem very obvious, it has its own share of disadvantages. First, the leader could run away with all the stake. Secondly, most networks have a minimum period of stake holding. This would greatly reduce the number of block proposals in which the nodes of mining pool participate.

Chapter 2

Challenges in Blockchain due to Mining pools

Since the launch of mining pools, a few centralised mining pools have dominated blockchain mining. In reality, the top ten mining pools presently control more than 95% of the hashing power. Pools can also control which transactions are included in the blockchain, possibly raising the risk of transaction censorship or preferring their own blocks, and can potentially coordinate their attempts to carry out a successful 51% attack. Furthermore, they can be a single point of failure in the event of a server failure or a denial-of-service attack. Apart from the security concern due to the centralization of mining pools, there are two primary attacks against the mining pools:

1) A **Block withholding** attack occurs when a miner infiltrates a mining pool in order to decrease its profits by distributing just the pPoW and not the fPoW. The pool operators mostly carry out this attack on its competing pool, infiltrating as a miner and then distributing the work among its miners, then sending back the shares to the other pool when pPoW is received, but when fPoW is received, it refrains/delays in sending the block back to the other pool, sabotaging its rewards.

2) **Pool-Hopping** attack, here the miner hops from one pool to another, mining only when

the attractiveness of the mining pool is high. The attractiveness of mining is measured in terms of expected earnings, variance, and maturity time, which varies according to the pool's current state primarily determined by the pool's hash rate.

Chapter 3

Proposed MODEL

3.1 Working

In order to counteract the dominance of centralized authority (pool), operator), we propose a democratic and semi-decentralized system.

Nodes are classified into three types: *common nodes*, *candidate nodes* as well as a *leader nodes*. By default, all mining nodes that join the mining pool consist of common nodes. In order for a node to be eligible to become a candidate node it must have the necessary qualifications. Hardware for full-node operation, and computation power above a threshold, (so that the node can function as a full node) is a criterion for inclusion in the consensus group. Then all these nodes take part in the election process and put out their agendas (primarily the payout scheme used and transaction fees charged by the mining pool) to the community and ask for votes. All the common nodes take part in stake-based voting and vote for their favorite candidate using their stake. Stake-Based voting means that the vote value of a common node is proportional to its stake. After each block cycle, the top n common nodes, on the basis of total accumulated stake-based votes, are elected as candidate nodes to the consensus group. These elected nodes interact with the blockchain network and with the common nodes of the mining pool.

Out of all the selected nodes, a leader node is selected every block cycle, which writes

the transactions to the block template. To select the leader node, all the candidate nodes possess a function that takes as input the current blockchain state and the stake-based vote count of all the candidate nodes and generates a pseudo-random sequence of candidates (called the leader sequence). This leader sequence is consistent with all the candidates, and candidates with more stake-based votes will have a higher probability of acquiring more positions. After every block cycle, the next candidate node in the sequence becomes the leader. If the candidate node which has to be made leader is faulty, then the next candidate node becomes the leader node. The responsibilities of all the nodes are discussed later.

3.2 Functions

The functions of all the nodes are mentioned here:

- Common Nodes:
  - Requesting block template from the leader or one of the candidate nodes.
  - Finding a nonce for the given transactions satisfying the mining pool threshold.
  - Sending its share to all candidates/leader nodes.
- Candidate Nodes:
  - Sending their local ledgers to the leader node
  - Getting the block template from the leader node and verifying all the transactions.
  - Providing the common nodes with the block template
  - Maintaining a local ledger of the shares contributed by the common nodes.
- Leader Node:
  - Maintaining a local ledger of contributions of each node, in order to efficiently distribute rewards.

- Constructing the block template.
- Sending the block template to all candidate nodes for verification.

3.3 Algorithm

Our mining pool algorithm essentially consists of 6 simple steps repeating after each block is mined which are explained below:

- **Step 0:** The leader node for the current block is decided using the pseudo-random sequence. The pseudo-random sequence generator takes as input the vote percentage of each candidate node. The expected number of times a candidate node appears in the sequence is proportional to its vote percentage.

- **Step 1:** All candidate nodes send their local ledgers to the leader node which updates its ledger using the majority of entries corresponding to each node in case of any discrepancies.

- **Step 2:** The leader node constructs the block template to be mined and sends it to all candidate nodes for verification.

- **Step 3:** If a majority of candidate nodes don't receive the block template or find that the transactions are not appropriate the leader gets penalized by disclosing this to all the nodes in the community and the next leader node is chosen according to the pseudo-random sequence. Else if the majority of candidate nodes verify all the transactions added to the block template using their local ledgers, the block template is accepted and stored by each candidate node for the future.

- **Step 4:** Common nodes receive the block template from any of the candidate nodes, then find a suitable nonce and send back the share to all candidate nodes so that each of them can update their local ledgers.

- **Step 5:** Upon finding a share satisfying the blockchain target the leader node adds the block to the blockchain network.

3.4 Achieving Democracy through the Proposed Model

Choosing the consensus committee through elections establishes democratic principles in the mining pool. All the candidate nodes present in the consensus group are elected by the common nodes and are publicly known to everyone (unlike the manager of a centralized mining pool). So they have to cater to the need of the community and stay away from malpractices. If a candidate node is found guilty of malicious behavior, then it will lose the confidence of common nodes. Then the common nodes can give their vote to some other candidate. If the malicious candidate node's total stake-based vote decreases substantially and is not among the top n candidates, it will be removed from the consensus group and, therefore, suffer the loss of the management fee it was receiving. In this way, the small miners will also have a say in the functioning and policy making of the pool. The elected consensus group will decide on any policy change like the payout system and threshold value.

Chapter 4

Code

We studied the implementation of Monero Pool to work upon. Here is the basic functioning of a Monero mining pool

- Initialize pool parameters
  - Set the pool fee
  - Set the minimum payout threshold
  - Set the pool wallet address
  - Set the mining pool difficulty
- Connect to the Monero network
  - Connect to the Monero daemon
  - Set the block template
- Receive and validate shares
  - Receive shares submitted by miners
  - Verify that the share meets the pool difficulty
  - Assign a share value to the share based on its difficulty

  - Accumulate the share values for each miner

- Distribute rewards
  - Calculate the total pool hash rate
  - Determine each miner's share of the pool's total hash rate
  - Calculate each miner's reward based on their share of the pool's hash rate and the block reward
  - Deduct the pool fee from each miner's reward
  - If a miner's accumulated rewards exceed the minimum payout threshold, pay out the rewards to the miner's wallet address

- Update pool information
  - Update the block template with the latest block information
  - Broadcast the new block template to miners
  - Repeat steps 3 to 5 until a block is successfully mined.

4.1 Files

The functions used in all the files are mentioned here:

- bstack.c file : This file in a Monero pool implementation contains the implementation of a block template stack, which is a data structure used by the pool to store a set of block templates that are available for miners to work on. This file provides functions for creating, manipulating, and freeing the block template stack.

  The code in `bstack.c` starts with the definition of the `bstack_t` struct, which represents the block template stack. It contains fields for the current count of block templates in the stack `c` (the capacity of the `bstack`), the current number of block

templates in the stack `cc` (the current number of items in the `bstack`), the size of each block template `z` (the size of each item in bytes), the total number of block templates pushed onto the stack `n` (the total number of items that have been pushed to the `bstack`), the current index of the top block template `ni` (the index of the next item to be popped from the `bstack`), a function pointer to a block recycling function `rf` (a function pointer to a user-defined function that is called to recycle an item when it is dropped from the `bstack`), and a pointer to the memory buffer used to store the block templates `b` (a pointer to a buffer that holds the items in the `bstack`).

The `bstack_new()` function creates a new block template stack with the specified number of block templates (`count`), the size of each block template (`size`), and a recycling function (`recycle`) that is called when a block template is dropped from the stack. This function allocates memory for the stack and its buffer, and initializes its fields.

The `bstack_free()` function frees the memory allocated for the block template stack and its buffer. If a recycling function was provided, this function calls it on each block template in the stack.

The `bstack_push()` function adds a block template to the top of the stack. If the stack is full, the recycling function is called on the oldest block template in the stack before the new block template is added. If an item is provided, its memory is copied into the buffer; otherwise, the buffer is zeroed out.

The `bstack_drop()` function removes the top block template from the stack. If the stack is empty, this function does nothing.

The `bstack_top()` function returns a pointer to the top block template in the stack, or `NULL` if the stack is empty.

The `bstack_count()` function returns the number of block templates in the stack.

The `bstack_next()` function returns a pointer to the next block template in the stack,

starting from the top. This function is used to iterate over all the block templates in the stack.

The `bstack_reset()` function resets the stack index so that the next call to `bstack_next()` returns the top block template.

- Forkoff.c file : This contains a function named `forkoff` which is used to daemonize a process in Unix-like systems. Daemonizing a process means to detach it from the controlling terminal and make it run in the background as a background process. The function takes a string `pidname` as an argument which specifies the name of the file to which the PID of the daemonized process should be written. If `pidname` is not `NULL`, the function checks if a process with the same PID is already running by attempting to read the PID from the specified file. If such a process is found, the function terminates with an error message.

  The function forks the current process twice using the `fork()` system call. The first fork creates a child process which becomes the leader of a new session by calling `setsid()`. The second fork creates a grandchild process which becomes the actual daemon process. This double fork technique is used to ensure that the daemon process does not acquire a controlling terminal, which could lead to problems if the terminal is closed or disconnected.

  After the second fork, the PID of the daemon process is written to the specified file, if `pidname` is not `NULL`. Finally, the standard input, output, and error streams are closed, and new file descriptors are opened for them that point to `/dev/null` or are duplicated from the existing streams.

  Overall, the `forkoff` function ensures that the process is completely detached from the user's terminal and other processes, and is properly configured to run in the background.

- Growbag.c file : This contains header file for a growbag data structure, which is es-

sentially an array that can be dynamically resized as needed. Here are the function prototypes listed in the header file.

  - `gbag_new`: Initializes a new growbag with a given `count` of elements, each with a given `size` (in bytes). The `recycle` parameter is a function that will be called on an element when it is removed from the growbag, and the `moved` parameter is a function that will be called if the growbag is resized and its location in memory changes. `out` is a pointer to a pointer to a `gbag_t` struct that will be initialized.
  - `gbag_free`: Frees the memory associated with a growbag, including any elements stored in it.
  - `gbag_get`: Retrieves a new element from the growbag, or increases the size of the growbag if necessary. Returns a pointer to the new element, or `NULL` if it could not be allocated.
  - `gbag_put`: Puts an element back into the growbag, making it available for reuse. `item` should be a pointer to the element being returned.
  - `gbag_max`: Returns the maximum number of elements that the growbag can store.
  - `gbag_used`: Returns the number of elements currently in the growbag.
  - `gbag_find`: Searches the growbag for an element with a given `key`, using a comparison function specified by `cmp`.
  - `gbag_find_after`: Searches the growbag for an element with a given `key`, starting the search at the element specified by `from`.
  - `gbag_first`: Returns a pointer to the first element in the growbag.
  - `gbag_next`: Returns a pointer to the next element in the growbag, starting from the element specified by `from`. If `from` is `NULL`, returns the first element.

- Pool.c file : The `pool.c` file is part of the Monero mining pool software implementation. This file contains the source code for the main logic of the mining pool server.

The mining pool server is responsible for receiving mining jobs from the Monero daemon, distributing these jobs to connected miners, and collecting the submitted shares from these miners. The pool server then aggregates these shares and submits valid blocks to the Monero network, and distributes the resulting rewards to the miners based on their contribution to the pool.

The `pool.c` file contains the implementation of the `pool_main()` function, which is the main entry point for the pool server application. This function initializes the pool server, starts listening for incoming connections from miners, and handles the processing of mining jobs and submitted shares.

The `pool.c` file also contains other functions that handle the communication with the Monero daemon, the management of the connected miners, and the validation and aggregation of submitted shares.

The code defines several structs and enums used in the Monero pool implementation:

- `block_status`: An enum to represent the status of a block. A block can be locked, unlocked, or orphaned.
- `stratum_mode`: An enum to represent the mode of operation for the pool. It can be either normal or self-select.
- `msgbin_type`: An enum to represent the type of binary message sent between the pool and the miners.
- `hr_stats_t`: A struct to store hash rate statistics for an account.
- `config_t`: A struct to store the pool configuration settings.
- `block_template_t`: A struct to store the block template data for mining.
- `job_t`: A struct to store the job data for a miner.
- `client_t`: A struct to store the client data for a connected miner.
- `account_t`: A struct to store the account data for a connected miner.

- `share_t`: A struct to store the share data submitted by a miner.
- `block_t`: A struct to store the block data.
- `payment_t`: A struct to store the payment data for an account.

It also contains some macros:

The `JSON_GET_OR_ERROR` macro is used to retrieve a JSON object from a parent object, and if it does not exist or is not of the expected type, an error is sent.

The `JSON_GET_OR_WARN` macro is similar, but instead of sending an error, it logs a warning message.

The `INPLACE_TO_BIN` macro converts a hex string to binary in-place.

The `compare_uint64`, `compare_string`, `compare_block`, and `compare_share` functions are used for sorting and searching data in a database using the LMDB library.

- util.c file : This is a C code implementing various utility functions for handling binary and hexadecimal data, as well as variable-length integer encoding/decoding.

  The ‘is_hex_string‘ function checks whether a given string contains only valid hexadecimal characters, returning ‘-1‘ if the input string is empty, ‘-2‘ if it contains any invalid character, and ‘0‘ if it is a valid hexadecimal string.

  The ‘hex_to_bin‘ function converts a hexadecimal string into a binary representation, where each pair of hexadecimal digits corresponds to one byte in the binary representation. The input hexadecimal string is passed as a pointer ‘hex‘, with the length specified by ‘hex_len‘, while the resulting binary representation is stored in a buffer pointed to by ‘bin‘, with maximum size specified by ‘bin_size‘.

  The ‘bin_to_hex‘ function performs the reverse operation, converting a binary representation into a hexadecimal string. The input binary data is passed as a pointer ‘bin‘, with size specified by ‘bin_size‘, while the resulting hexadecimal string is stored in a buffer pointed to by ‘hex‘, with maximum size specified by ‘hex_size‘.

The ‘reverse_bin‘ function simply reverses the order of bytes in a binary buffer, where the input buffer is pointed to by ‘bin‘ with size specified by ‘len‘.

The ‘stecpy‘ function is a safe version of ‘strcpy‘, which copies characters from a source string ‘src‘ to a destination string ‘dst‘, up to the provided end pointer ‘end‘. It returns a pointer to the next available position in the destination buffer.

The ‘trim‘ function removes leading and trailing whitespace characters from a given string, returning a pointer to the start of the trimmed string.

The ‘read_varint‘ function decodes a variable-length integer encoded as a sequence of bytes, where the input bytes are passed as a pointer ‘b‘. It returns the decoded integer value as a ‘uint64_t‘.

The ‘write_varint‘ function encodes a given integer value ‘v‘ as a variable-length integer and writes the resulting bytes to a buffer pointed to by ‘b‘. It returns a pointer to the next available position in the output buffer after writing the encoded bytes.

- Webui.c file : This code is an implementation of a web user interface (webui) for a mining pool. The webui serves the pool's statistics and workers' information in JSON format via HTTP requests. The implementation uses the libevent library for event-driven network programming and pthread library for multithreading.

  The code starts by including necessary headers, including standard C libraries, libevent headers, and custom headers for logging and the mining pool implementation. The implementation defines a global variable "handle" of type pthread_t, "webui_base", "webui_httpd", and "webui_listener" of type event_base, evhttp, and evhttp_bound_socket, respectively. It also defines a global constant "webui_html" and "webui_html_len" for the HTML content of the webui.

  The implementation then declares a static function "fetch_wa_cookie," which takes an HTTP request and returns the "wa" cookie value if it exists in the HTTP request. The "wa" cookie is used to identify the worker that made the request.

The implementation declares two static functions, "send_json_workers" and "send_json_stats," which respectively send the list of workers and pool statistics in JSON format via HTTP responses. Both functions take an HTTP request and a void pointer as arguments. The void pointer is used to pass the pool statistics to "send_json_stats" function.

The "process_request" function is declared static and takes an HTTP request and a void pointer as arguments. It checks the URL of the HTTP request and calls either "send_json_workers" or "send_json_stats" function to send the appropriate JSON response.

The main function initializes the event-driven network programming library and starts the web server. It first initializes the logging system and the mining pool implementation. It then creates a new event_base using "event_base_new" function, creates a new evhttp object using "evhttp_new" function, and binds the evhttp object to a socket using "evhttp_bind_socket_with_handle" function. The HTTP server is set up to listen on port 8080. Finally, the main function enters an event loop using "event_base_dispatch" function to handle incoming HTTP requests.

- XMR file: This file containst the C++ implementation of Monero's hash function, and some other utility functions. The functions include:

    1. `nettype_from_prefix`: a function to convert a prefix to a network type.
    2. `get_hashing_blob`: a function that computes the hashing blob for a given block blob.
    3. `parse_address`: a function that decodes a Monero address into its components.
    4. `is_integrated`: a function that determines whether a given prefix is for an integrated address.
    5. `get_block_hash`: a function that computes the block hash for a given block blob.
    6. `get_hash`: a function that computes the hash of a given input.

7. `set_rx_main_seedhash`: a function that sets the main seed hash for the RandomX hash function.
8. `get_rx_hash`: a function that computes the RandomX hash of a given input.
9. `validate_block_from_blob`: a function that validates a block blob.

4.2 Algorithm for voting

We need to conduct peer-to-peer voting because of the decentralized nature of the pool. Before casting vote, a pseudo-random sequence of candidate nodes is to decide on the leader node for the current block cycle. Here is the sample code for generating pseudo-random sequence:

```
typedef struct {
    int id;
    double weight;
} Node;

typedef struct {
    int node_id;
    int vote;
} Vote;

int num_nodes;
Node nodes[MAX_NODES];
Vote votes[MAX_VOTES];
int num_votes = 0;

void add_node(int id, double weight) {
```

```
    nodes[num_nodes].id = id;
    nodes[num_nodes].weight = weight;
    num_nodes++;
}

void cast_vote(int node_id, int vote) {
    votes[num_votes].node_id = node_id;
    votes[num_votes].vote = vote;
    num_votes++;
}

void send_vote(int dest_id, int vote) {
    // send vote to dest_id
    // ...
}

void receive_vote(int src_id, int vote) {
    // add vote to local list of votes
    cast_vote(src_id, vote);
}

void conduct_vote() {
    int num_votes_per_node[num_nodes];
    memset(num_votes_per_node, 0, sizeof(num_votes_per_node));
    for (int i = 0; i < num_votes; i++) {
        int node_id = votes[i].node_id;
        int vote = votes[i].vote;
```

```
            num_votes_per_node[node_id]++;
            double total_weighted_votes = 0;
            for (int j = 0; j < num_nodes; j++) {
                total_weighted_votes += nodes[j].weight * num_votes_per_node[j];
            }
            if (total_weighted_votes >= 0.5 * num_votes) {
                printf("Consensus reached!\n");
                break;
            }
        }
    }
```

In the above code, the Node struct represents a node in the network, and contains an id and a weight value. The Vote struct represents a vote, and contains a node_id and a vote value. The add_node() function adds a node to the list of nodes, and the cast_vote() function adds a vote to the list of votes. The send_vote() and receive_vote() functions simulate sending and receiving votes between nodes.

The conduct_vote() function computes the total weighted votes for each node, and checks if a consensus has been reached (i.e., if the total weighted votes for any node is greater than or equal to half of the total number of votes). If a consensus is reached, the function prints a message indicating that a consensus has been reached.

The following is the code for conducting elections:

1. Each node initializes its own tally to 0.
2. Each node generates its own result.
3. Each node broadcasts its result to all other nodes in the network.
4. Each node receives all the results from other nodes.

5. Each node updates its own tally by adding the (stake) of the other nodes' results.

6. Each node broadcasts its updated tally to all other nodes in the network.

7. Each node receives all the updated tallies from other nodes.

8. Each node checks if the maximum tally is from itself. If so, it declares itself as the winner and broadcasts the result to all other nodes. If not, it waits for the next round of voting.

9. The election process ends when a node is declared as the winner.

```
// Step 1: Generate random number and send to all other nodes
for i in range(num_nodes):
    if i == node_id:
        continue
    random_number = generate_random_number()
    send_message(i, random_number)

// Step 2: Receive random numbers from all other nodes
received_numbers = []
while len(received_numbers) < num_nodes - 1:
    message = receive_message()
    sender_id, number = extract_message_info(message)
    if sender_id != node_id:
        received_numbers.append(number)

// Step 3: Calculate own vote
own_vote = calculate_vote(generated_number, received_numbers)
```

```
// Step 4: Send own vote to all other nodes
for i in range(num_nodes):
    if i == node_id:
        continue
    send_message(i, own_vote)

// Step 5: Receive votes from all other nodes
received_votes = []
while len(received_votes) < num_nodes - 1:
    message = receive_message()
    sender_id, vote = extract_message_info(message)
    if sender_id != node_id:
        received_votes.append(vote)

// Step 6: Calculate final election result
final_result = calculate_election_result(received_votes)

// Step 7: Broadcast final election result to all other nodes
for i in range(num_nodes):
    if i == node_id:
        continue
    send_message(i, final_result)
```

Chapter 5

Conclusion and Future Work

Our model is an attempt to decentralize the mining pools to the extent that is required. With the committee based consensus, we are removing single point of failure. If a candidate node is faulty, then others can keep the mining pool up and running for the time being. This doesn't leave the common miners to the mercy of the manager. Because of the representative democracy employed in the mining pool, the grievances and aspirations of all the miners are taken care of. If a candidate node comes up with policy changes in favour of the common nodes, it would be elected to the consensus committee. If any candidate node doesn't take care of the interests of the common miners, then it can be removed from the consensus committee. This is similar to the real life democracy.Apart from our model we discussed the functioning of mining pools and the security issues they pose. We also looked into the various PoS based consensus mechanism. Some of functionalities and structure of our model are derived from them, for example the pseudo random sequence from the committee based PoS and elected consensus group was derived from Delegate Pos. We concluded that normal PoW based mining pool cannot function in the PoS based blockchain network. Structure and policies of the mining pool need to be upgraded in order to be useful for PoS based blockchain network.

Currently our model contains a lot of broadcast messages within the mining pool nodes but due to limitations of both time and resources we couldn't test our model for efficiency and network bandwidth requirements. We can deploy smart contracts for the agendas by the candidate nodes that will automatically get executed if the candidate node wins a policy change election. With the current model, the candidate node might cheat by changing the policy later on but using smart contracts removes this threat. The pseudo-random sequence used to introduce decentralization by changing the leader on each block can be replaced by a consensus mechanism. Instead of one leader doing the block proposal at a time, a consensus mechanism could be introduced where all the candidate nodes are involved in deciding which transactions must be included in the block template.