Module Code: CS3BC20
Assignment report Title: Blockchain Coursework Assignment
Student Number: 29021144
Date (when the work completed): 03/03/2024
Actual hrs spent for the assignment: 30

Project Setup

The project setup uses windows forms to create an intractable UI where the user can interact with the mini blockchain application as shown in Figure 1. Event handlers were implemented to establish dynamic interaction between users and the blockchain interface which allows real-time responses to actions such as initiating transactions or mining new blocks.

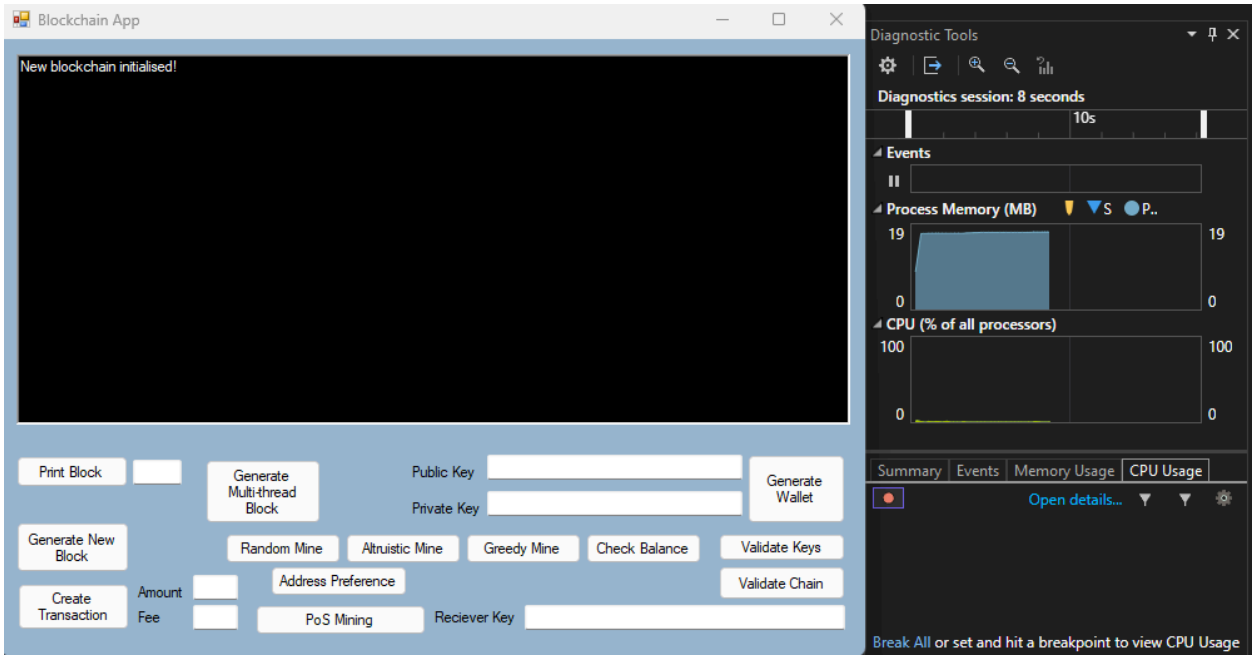

Figure 1, Project setup of blockchain app with interactable buttons

Blocks and the Blockchain

In the blockchain application, we created the Block and Blockchain classes, each block, serving as a fundamental unit within the blockchain, has a timestamp, an index, the previous block hash, a unique hash, a merkle root for transaction integrity, miner address, transaction list, nonce, and a reward system, forming the block structure. A constructor was implemented for the genesis block, as its unique properties and role in the blockchain. Hashing is a critical function within the system, utilising the SHA256 asymmetric function to ensure cryptographic security. Each block's hash is represented as a hexadecimal string in the user interface, demonstrating the application's basic functionality and the immutable nature of the blockchain as every block includes the hash of the previous block it is linked to the previous block and thus linked to the original genesis block. In Figure 2 looking at the appendix, you can see that when pressing ‘Generate Block’ a block is added to the blockchain and is output to the user interface.

Transactions and Digital Signatures

The Wallet class in the blockchain application is a crucial component for managing user transactions, including key pair generation, digital signature creation, and validation. It includes the essential functionalities required for a user to interact securely within the blockchain network.

The transaction class is crucial in managing transactions in the blockchain, variables such as timestamp, senderAddress, recipientAddress, amount, fee, hash and signature are used for their roles in transaction processing. The constructor initializes these variables and employs SHA256 hashing to generate a unique hash for each transaction, ensuring its immutability and uniqueness.
Key Pair Generation: Upon creation of a new Wallet object, an asymmetric key pair is generated using the Elliptic Curve Digital Signature Algorithm (ECDSA) with a P-256 curve. This process ensures the generation of a secure public and private key pair. The public key (publicID) is intended for open sharing within the network to identify the wallet, while the private key is typically kept secret for signing transactions to prove ownership of said wallet. The hash of these keypairs can be seen in Figure 3 when the Generate Wallet button is pressed.

The Digital signature is a pivotal aspect of this class where the sender's private key is used to sign the transaction hash. This process authenticates the transaction, linking it to the sender and preventing any alteration post-signature without invalidating the transaction. The signature ensures that the transaction has not been tampered with and verifies the sender's consent to the transaction details.

Finally, this blockchain includes a transaction pool which is a dynamic list of pending transactions awaiting confirmation in the blockchain. This pool is essential for managing transaction throughput and organizing the order of transaction processing, reflecting a realistic blockchain environment where multiple transactions are queued and processed over time. In Bitcoin and other blockchain systems, this is typically called the mempool and typically miners prioritise the transactions with the highest fees due to the economic cost of mining bitcoin. (GSR)
In summary, the Transaction class includes the core functionalities necessary for creating, signing, and managing transactions in the blockchain, Figure 4 shows what a transaction looks like on the blockchain.

Consensus Algorithms (Proof Of Work)
The Proof-of-Work (PoW) algorithm is used in a distributed environment to prevent gaming of the system and to ensure decentralisation, where a correct hash value is sought by altering the nonce until the hash meets the specified difficulty criteria. This algorithm underpins the security and decentralisation of the blockchain, although it comes with trade-offs like significant computation thus energy consumption in a large enough network and potential delays in transaction processing due to the mining difficulty.
Nonce generation ensures each mining operation has a unique starting point, which is essential for the PoW mechanism to function correctly, preventing any possibility of pre-computing future blocks and ensuring a fair and competitive mining process.
.
The mining difficulty directly dictates the number of leading zeros required in a block hash. The difficulty level is a representation of how computationally difficult it is to brute force a hash that meets the required criteria for adding a new block. The higher the difficulty, the more time/computational difficulty is required to find a valid hash. In this blockchain implementation,

the difficulty is initially set at 4 to aim for an average block time of under 10 seconds on my machine.

Rewards and fees are used to incentivize miners to contribute to network security through the mining process. The reward, coupled with transaction fees, motivates miners to continue their computational efforts to validate transactions and generate new blocks, thereby securing the blockchain and facilitating transaction processing. In my blockchain implementation, the reward is set to 100 coins for the successful miner as seen in Figure 2.

Validation

The Blockchain class maintains the integrity of the blockchain network via several core functionalities essential for validating the blockchain's structure and coherence between blocks and managing the transaction and block validation processes.

Block Coherence and validation checks: The blockchain class maintains a list of blocks, ensuring that each block is linked to the previous one through its hash value. This linkage forms a continuous chain, where altering any previous block would invalidate the entire chain after that. These validation checks are crucial to the integrity and trustability of the network as it makes tampering obvious and computationally impractical. By pressing the `Validate Chain` button the user can see if the blockchain has been altered as seen in Figure 5.

Checking and Validating Balances (Ledger Tracing): The `GetBalance` method allows tracing the transaction history associated with a wallet address, by summing credits and debits of a wallet to determine the current balance as seen in Figure 6. This tracing combats misuse by ensuring that all transactions are accounted for and that no funds can be double-spent or fabricated, reinforcing transparency and security. The Public and private key can also be validated by pressing the `Validate Keys` button as shown in Figure 7 when the key fields have data present.

Validating Blocks and Merkle Roots: The Blockchain class includes functionality to validate each block's integrity through hash validation (ValidateHash) and Merkle root validation (ValidateMerkleRoot). The Merkle root is a single hash that represents all transactions in a block, giving a compact way to verify the set of transactions without needing to check individually. This contributes to the blockchain's efficiency and security, ensuring that the transaction data within each block remains unchanged.

The blockchain includes methods to continuously verify the correctness of its data, such as checking if each block's hash and Merkle root are valid. These verification steps are for maintaining the blockchain's reliability and trustworthiness, ensuring that each component adheres to the defined "rules" of the system, thereby preserving the integrity and security of the entire blockchain network.

Task 1 - Extending Proof-of-Work

By integrating multi-threading techniques into the PoW process there are significant efficiency gains which are clearly visible in Table 1. In Figure 9 it is observed that all 24 logical cores of the processor were being utilised to mine the next block. The code for this multi-threaded PoW method can be found in the appendix, Figure 10.

Difficulty	Threads	n=1 (ms)	n=2 (ms)	n=3 (ms)	n=4 (ms)	$\bar{X}$ (ms)
6	4	7686	6976	67312	139802	55444
6	24	4949	3637	4319	1162	3516.75
5	4	2391	1253	1647	2926	2054.25
5	8	700	831	492	1571	898.5
5	16	1607	1208	2885	755	1613.75
5	1	11085	4464	4028	7939	6879
3	24	7	37	22	3	17.25
3	1	32	15	39	19	26.25

Table 1, A comparative study between thread count and mining times

Volatility in block times is evident where a block mined at difficulty 6 on 4 threads can take 139802ms to find the solution while with the same parameters, a block could take 7686ms to mine emphasising the variability of block times in blockchain systems like this mini blockchain and in systems like bitcoin where some blocks can take 122 minutes to mine according to (u.today).

The mining process in Figure 10 is split across multiple tasks/threads, each running in parallel and attempting to find a valid hash by iterating through a unique range of nonce values for each task/thread. A CancellationToken is used to stop all tasks as soon as any thread finds a valid hash, optimising resource usage and minimizing computation time. The CreateHashWithNonce method is introduced to handle hash creation for a given nonce. Once a task finds a valid hash, it sets a shared flag (hashFound) to signal other tasks to stop their execution, overcoming the duplication of work problem common in parallelised systems.

Task 2 - Adjusting Difficulty Level for Proof-of-Work

The Difficulty adjustment mechanism aims to converge average block mining times towards a target block time. The code uses an adaptive difficulty algorithm that dynamically adjusts the difficulty to produce a block about every 10 seconds in this instance. The system measures blocktime by recording the duration taken to mine each block using a Stopwatch instance. These times are stored in `List<double> miningTimes`, which keeps track of mining times to

calculate the moving average. This moving average is then used to determine whether the difficulty should be adjusted to achieve the target block time.
If the average mining time is greater than the target, indicating that blocks are being mined too slowly, the difficulty is decreased. Otherwise, if the average mining time is less than the target, suggesting that blocks are being mined too quickly, the difficulty is increased. This adjustment is made to keep the block time close to the target, ensuring consistency in block generation rates regardless of fluctuations in network hash power. This Difficulty adjustment in the blockchain system can be observed in Figure 11 and Figure 12.
This implementation is justified by the need to maintain consistent block times across the blockchain network. This consistency is vital to ensure predictable transaction processing times and network stability. Thus allowing the network to respond to changes in mining power, ensuring that the blockchain remains secure and functional regardless of external factors. The choice of a 10-second block time reflects a balance between transaction throughput and network stability.

Task 3 - Implementing Alternative Mining Preference Settings

The transaction selection options are visible in Figure 1 and allows the user to adapt the transaction selection strategy for transactions to be included in the block. This is crucial for simulating real-world blockchain scenarios where miners prioritise different transactions.
The greedy selection strategy prioritizes transactions with the highest fees. This strategy aims to earn the miner the most possible by sorting the transactions by the highest fee. The code is visible in Figure 13. In this scenario, users are willing to pay more for quicker transaction processing to have their transaction included in the next block.
The Altruistic strategy selects transactions based on their waiting time, prioritising those that have been pending for the longest time, this approach is fair and ensures all users' transactions are included regardless of the fee. This strategy could be adopted by a miner who is less focused on immediate rewards.
The Random selection selects transactions at random providing all pending transactions an equal chance of being included in the next block. This strategy can be used to simulate an environment where no transaction policy dominates over the network.
Address Preference prioritises transactions based on the senderAddress, which could be used to favour transactions that could exhibit illicit or coerced behaviour within the network.

Task 4 - PoS Implementation

The PoSBlock class introduces unique premises such as Validators and Staking, A validator stakes an amount to participate in the block creation process and is represented as ValidatorAddress. The constructor of the PoSBlock class takes parameters from the last block, a list of transactions, the validator's address, and the staked amount. It initializes the block by calling the base class constructor and then invokes the MinePoS method to simulate the mining process specific to PoS. The MinePoS method demonstrates the core logic of PoS mining. Unlike PoW, which requires computational work to solve cryptographic work, PoS mining is a basic selection process where validators have a chance to create a block based on their stake. The randomNumber simulates the randomness factor in the selection process. If the generated

random number is less than the ratio of the validator's stake to the total stake, the validator is allowed to create the block.
Proof of stake reflects a probabilistic method of achieving consensus, emphasizing the role of validators' economic commitment to the network's integrity and doesn't require as much computational work as PoW which can be seen as a large benefit. However, some downsides to PoS include validator centralisation and collusion risk (ImmuneBytes).

References:

GSR.(n.d.). Chart of the Week: Bitcoin Mining Part 1 - Overview of a Transaction. From: https://www.gsr.io/reports/chart-of-the-week-bitcoin-mining-part-1-overview-of-a-transaction

u.today. (2021, April 19). Bitcoin Just Recorded Longest Time Between Two Blocks in Almost 10 Years. u.today. https://cryptonews.net/news/bitcoin/553800/

ImmuneBytes.(n.d.) The PoS Timebomb in Blockchain Networks. https://www.immunebytes.com/blog/the-pos-timebomb-in-blockchain-networks/

Appendix:

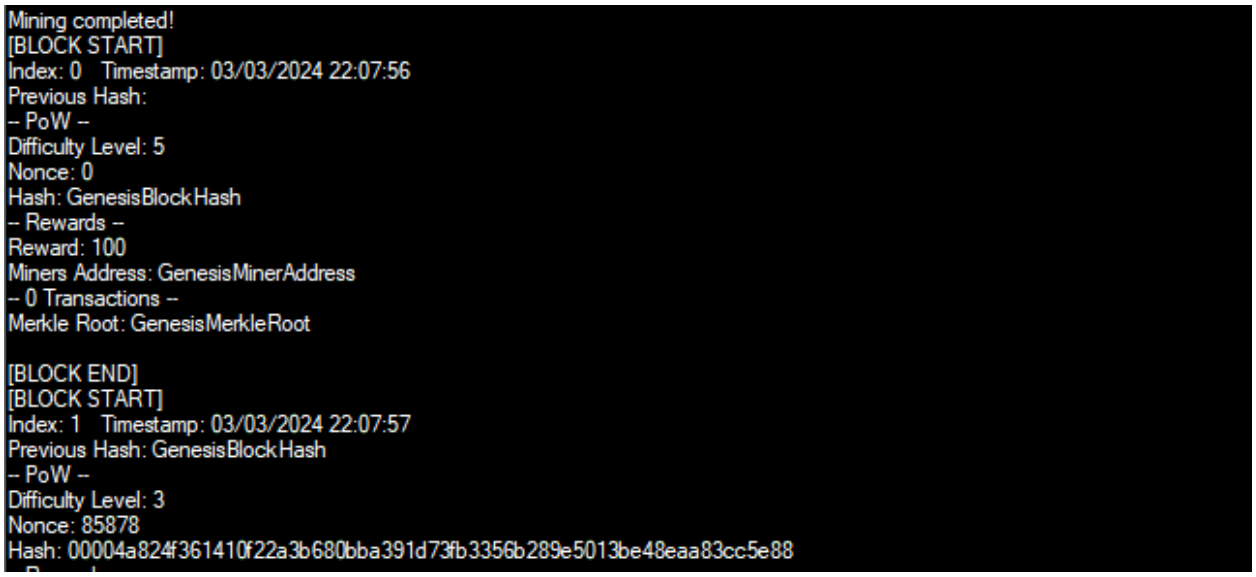

Figure 2, Generating a new block

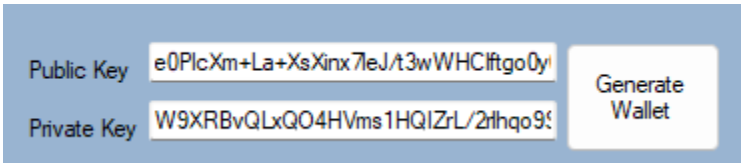

Figure 3, Generating wallet public private key pair

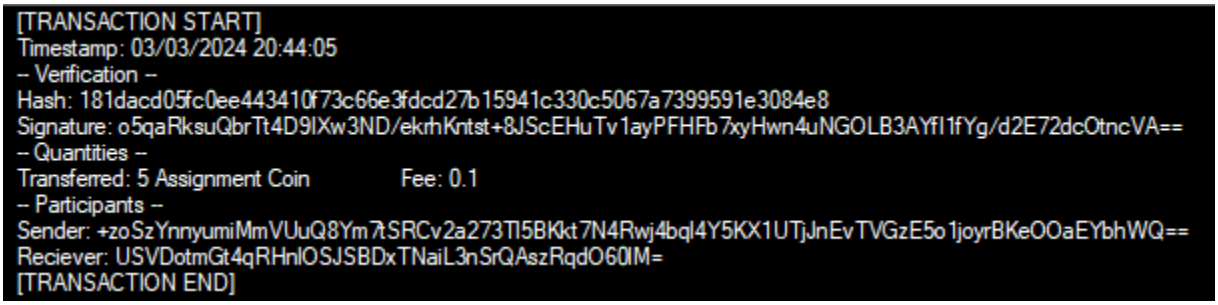

Figure 4, Inside a transaction

Figure 5, Validation of the Blockchain

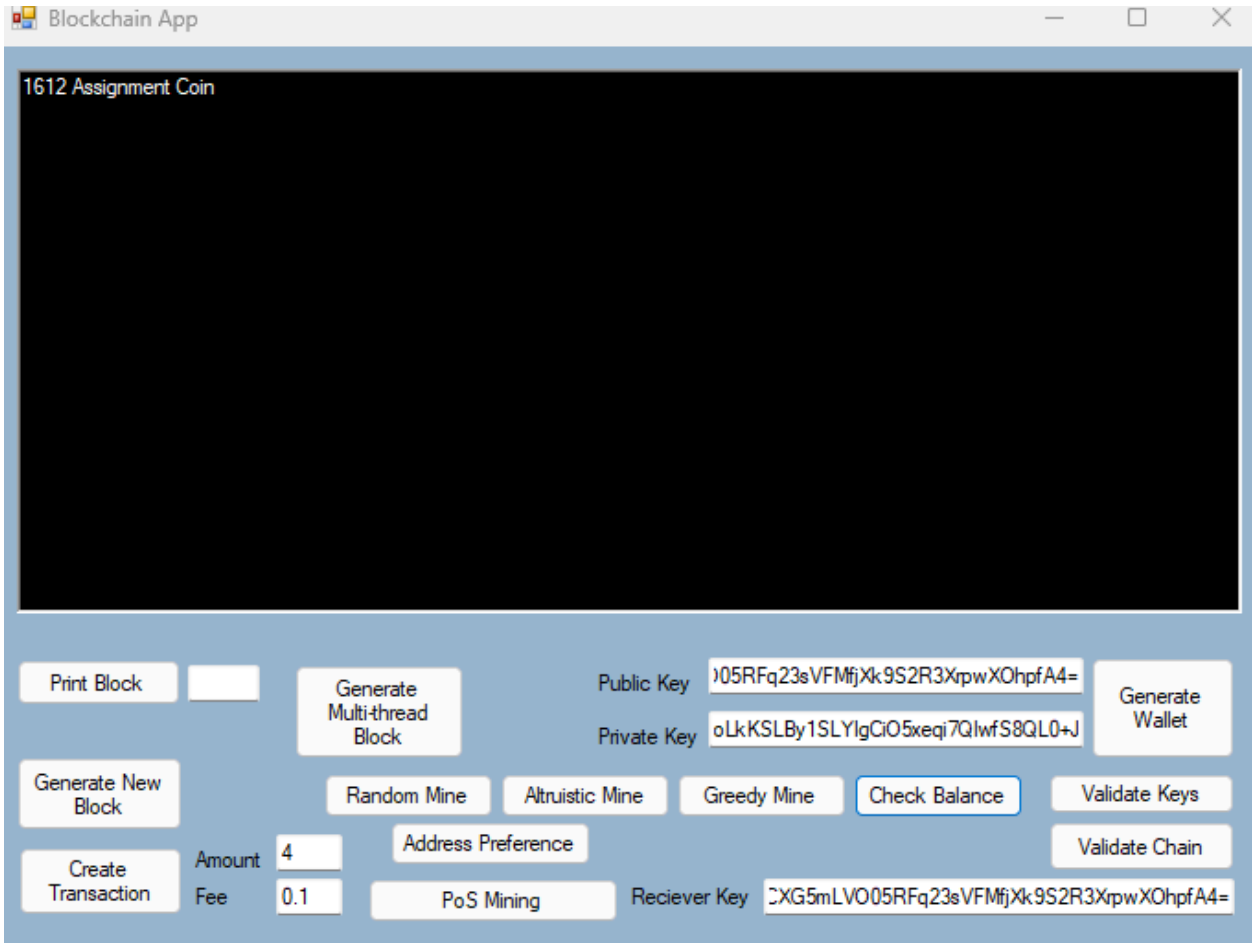

Figure 6, Checking the balance of a wallet.

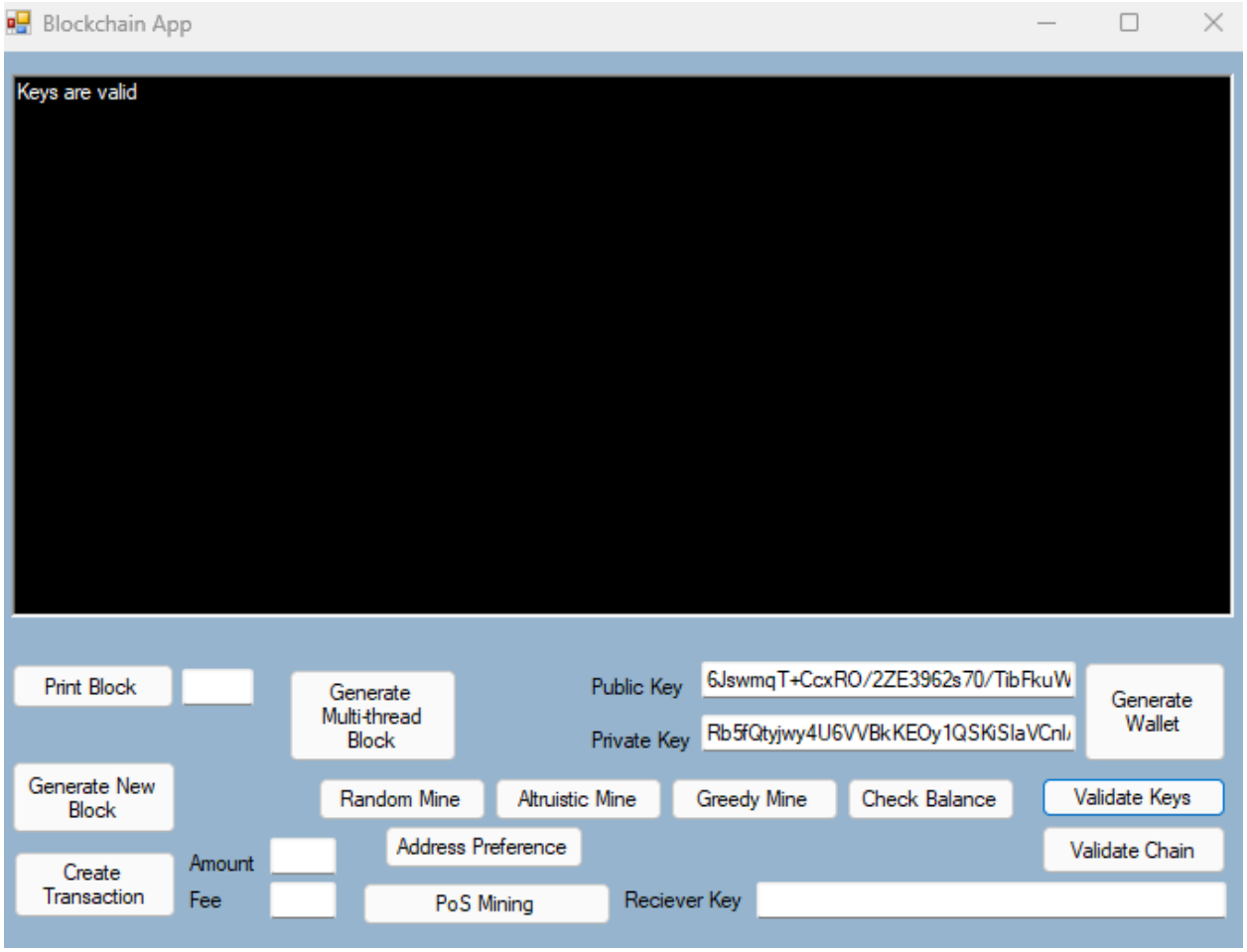

Figure 7, Checking if the public and private keys are valid

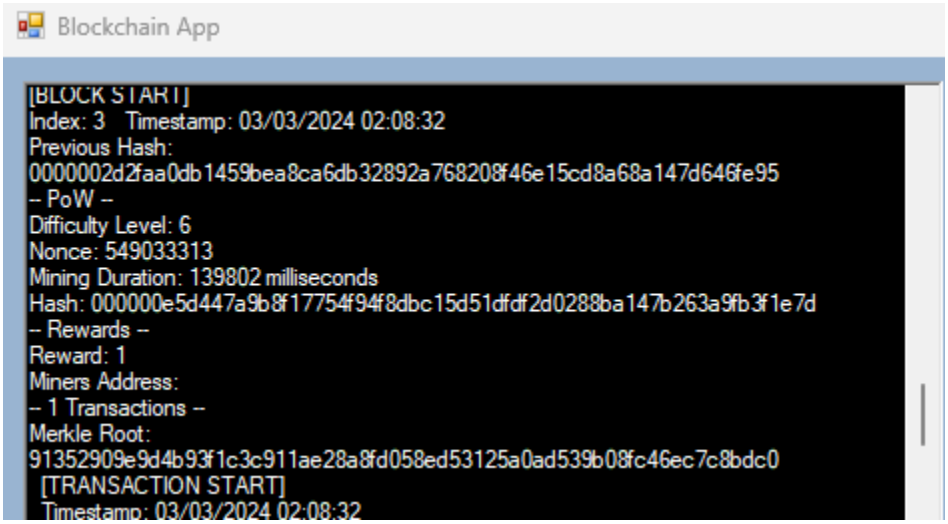

Figure 8, a rare Block taking 139802ms to be mined on difficulty 6 with 4 threads

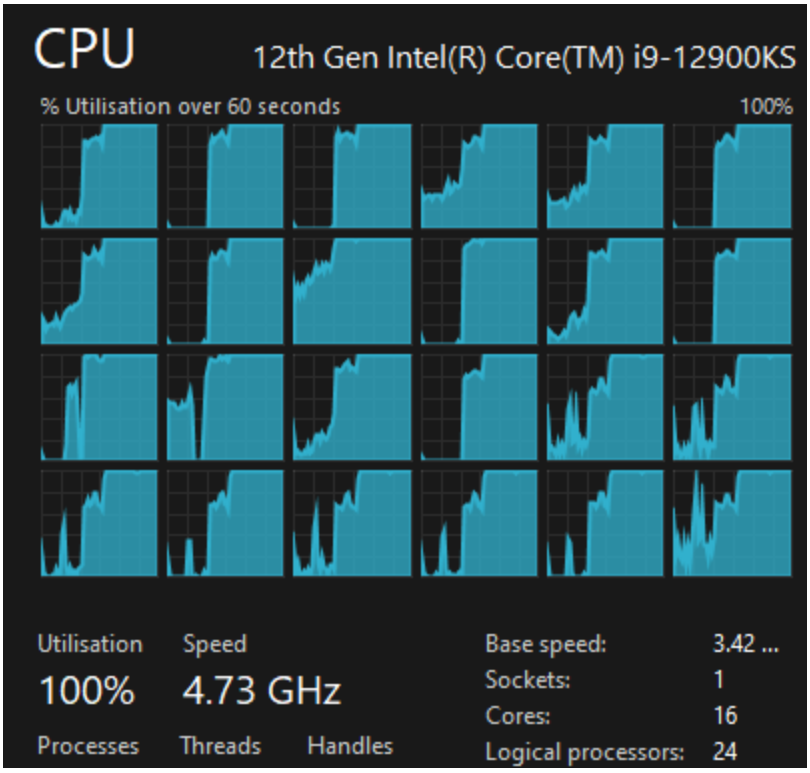

Figure 9, All logical processors being utilised to mine

```
public String Mine()
{
    Stopwatch stopwatch = Stopwatch.StartNew();
    bool hashFound = false;
    string validHash = null;
    long validNonce = 0;

    // 24 threads on my pc
    int numberOfThreads = Environment.ProcessorCount;
    var tasks = new List<Task>();
    // Define a cancellation token
    CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
    CancellationToken token = cancellationTokenSource.Token;
    for (int i = 0; i < numberOfThreads; i++)
    {
        var task = Task.Run(() =>
        {
            long threadNonce = GetStartingNonce(Task.CurrentId.HasValue ? Task.CurrentId.Value : 0);
            string threadHash;
            while (!hashFound && !token.IsCancellationRequested)
            {
                threadHash = CreateHashWithNonce(threadNonce);

                if (threadHash.StartsWith(new string('0', difficulty)))
                {
                    lock (nonceSemaphore)
                    {
                        if (!hashFound)
                        {
                            hashFound = true;
                            validHash = threadHash;
                            validNonce = threadNonce;
                            cancellationTokenSource.Cancel();  // Stop all other threads
                        }
                    }
                }
                threadNonce++;
            }
        }, token);
        tasks.Add(task);
    }
    Task.WaitAll(tasks.ToArray());
    stopwatch.Stop();
    nonce = validNonce;  // Update the nonce with the valid value found by the threads
    Console.WriteLine($"Block mined with hash: {validHash}");
    Console.WriteLine($"Mining time: {stopwatch.Elapsed.TotalSeconds}s");

    return validHash;
}
```

Figure 10, Code for multi-threaded mine() function

```
-- PoW --
Difficulty Level: 5
Nonce: 1521213255
Mining Duration: 2268 milliseconds
Hash: 0000038f5d67aea5f8d43e41f6bc1d8cfb7f5b1bb9d998e1ffb30af02f2e3b51
-- Rewards --
Reward: 0
Miners Address:
-- 0 Transactions --
Merkle Root:

[BLOCK END]
[BLOCK START]
Index: 1    Timestamp: 03/03/2024 13:07:47
Previous Hash: 0000038f5d67aea5f8d43e41f6bc1d8cfb7f5b1bb9d998e1ffb30af02f2e3b51
-- PoW --
Difficulty Level: 6
```

Figure 11, Difficulty increasing from 5 to 6 after a Difficulty Adjustment

```
private void AdjustDifficulty()
{
    // Calculate moving average
    double movingAverage = miningTimes.Take(10).Reverse().Average();

    // Adjust difficulty based on moving average
    if (movingAverage < targetBlockTime && difficulty > 1)
    {
        difficulty--;
    }
    else if (movingAverage > targetBlockTime)
    {
        difficulty++;
    }
}
```

Figure 12, AdjustDifficulty() method

```
private void SortTransactions(List<Transaction> transactions)
{
    switch (selectionStrategy)
    {
        case TransactionSelectionStrategy.Greedy:
            transactions.Sort((t1, t2) => t2.fee.CompareTo(t1.fee)); // Sort by highest fee first
            break;
        case TransactionSelectionStrategy.Altruistic:
            transactions.Sort((t1, t2) => t1.timestamp.CompareTo(t2.timestamp)); // Sort by longest wait first
            break;
        case TransactionSelectionStrategy.Random:
            transactions.Shuffle(); // Shuffle the transactions randomly
            break;
        case TransactionSelectionStrategy.AddressPreference:
            transactions.Sort((t1, t2) => t1.senderAddress.CompareTo(t2.senderAddress)); // Sort by sender address preference
            break;
    }
}
```

Figure 13, Transaction Selection Code

```
----------PoS----------
Difficulty Level: 5
Nonce: 1252590
Hash: 00000b2761d8c768a02b0ff1ed97a5d20b75a9e389a08301b994b02d1b2c6b97
-- Rewards --
Reward: 100
Miners Address: Validator1
-- 1 Transactions --
Merkle Root: f43f3d48b2ccf6a4b790dded18aefffd2b37db75df8e2756df96d14a37753376
 [TRANSACTION START]
 Timestamp: 04/03/2024 02:59:27
 -- Verification --
 Hash: eea339744e9a56148f0862de27bb5c4b9c91e661c2ae963df64755a249204527
 Signature: zuMhD4ER4UOyrw+lEvS3tLZGMmFle5tok/6whjo0eFRlugGSyB3WToMTgQ/fzV5nqEs0Yz5LCXVqu7LK1iZP4w==
 -- Quantities --
 Transferred: 100 Assignment Coin        Fee: 0
 -- Participants --
 Sender: Mine Rewards
 Reciever: Validator1
 [TRANSACTION END]
[BLOCK END]
```

Figure 14, Proof Of Stake Block creation

```
{
    public class PoSBlock : Block
    {
        public double StakedAmount { get; set; }
        public string ValidatorAddress { get; set; }

        public PoSBlock(Block lastBlock, List<Transaction> transactions, string validatorAddress, double stakedAmount)
            : base(lastBlock, transactions, validatorAddress)
        {
            ValidatorAddress = validatorAddress;
            StakedAmount = stakedAmount;
            MinePoS();
        }
        private void MinePoS()
        {

            Random random = new Random();
            double randomNumber = random.NextDouble();

            // The validator is selected based on their staked amount.

            if (randomNumber < (StakedAmount / 10000.0)) // Assuming a total stake in the system is 10,000 for simplicity
            {
                string combinedData = $"{timestamp}-{prevHash}-{ValidatorAddress}-{StakedAmount}";
                hash = CalculateHash(combinedData);
                Console.WriteLine($"PoS Block mined by {ValidatorAddress} with stake {StakedAmount}");
            }
            else
            {
                Console.WriteLine($"Validator {ValidatorAddress} with stake {StakedAmount} was not selected to mine the next block.");
                // validator 2
            }
        }


        private string CalculateHash(string input)
        {
            using (SHA256 sha256 = SHA256.Create())
            {
                byte[] hashBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(input));
                return BitConverter.ToString(hashBytes).Replace("-", "").ToLower();
            }
        }
    }
}
```

Figure 15, PoS Code

Gitlab:
https://gitlab.act.reading.ac.uk/ih021144/29021144CS3BC20