Module Code: CS3BC  
Assignment report Title: Blockchain Coursework Assignment  
Student Number: 30002734  
Date (when the work was completed): 12/05/2025  
Actual hrs spent on the assignment: 30

Part 1

The solution file BlockchainAssignment.sln was opened in Visual Studio, and BlockchainApp.cs was launched in the Windows Forms designer. A single-line TextBox, a Button and a large RichTextBox were added from the Toolbox. In the Properties panel, the TextBox was renamed to inputText, the Button to printButton with its Text property set to “Print,” and the RichTextBox to outputBox so that each control’s purpose was immediately clear. In the code view, the Print button’s Click event handler was generated automatically, and a single line of code was inserted to assign the contents of inputText to outputBox.Text. After resizing and anchoring the controls for a clean layout, the application was run, and the word “test” was entered into inputText before clicking Print. The same text appeared instantly in outputBox, confirming that control placement, naming conventions, and event handling were functioning correctly as shown in the appendix.

Part 2

The core blockchain logic was implemented by creating two new classes, Block and Blockchain. In Block.cs, private fields for timestamp, index, previous hash and hash were declared. A parameterless constructor was provided for the genesis block that sets the timestamp to the current time, the index to zero and the previous hash to an empty string before calling CreateHash to compute its SHA-256 hash. The CreateHash method concatenates the index, timestamp and previous hash into a single string, runs it through the SHA256Managed hasher and formats the resulting 32-byte digest as a lowercase hexadecimal string where each byte is two hex characters to become the block’s hash. Overloaded constructors were also added to accept the last block so that new blocks automatically inherit the correct index and link back to their predecessor before computing their own hex-encoded hash.

In Blockchain.cs, a List of Block objects was declared and initialised in the constructor with a single new Block so that the chain always begins with the genesis block. A method named getBlockAsString was introduced to return the ToString representation of any block by index or to return an error message if the index is out of range. In the form’s constructor in BlockchainApp.cs, the existing UI controls were retained, and immediately after InitializeComponent, a new Blockchain object was instantiated. Calling getBlockAsString(0) replaced the initial console text with the details of the genesis block. When the application runs, it now displays the block index, timestamp, SHA-256 hash and previous hash (empty for block zero) in the RichTextBox, confirming that block creation and chaining behave as intended.

The appendix includes two images. The first shows the RichTextBox immediately after initialization with the “New blockchain initialised!” message. The second shows the running application displaying the full details of block zero in the RichTextBox.

Part 3

In the next phase, full wallet support and transaction creation were added to enable secure transfers of Assignment Coins. The application leverages the Wallet class, which implements ECDSA key‐pair generation and validation. Two new UI buttons, Generate Keys and Validate Keys, were placed beside text boxes labelled Public Key and Private Key. Clicking Generate Keys calls Wallet’s constructor, fills the Public Key field with the newly created public identifier and writes the matching private key into the Private Key field. Clicking Validate Keys then invokes Wallet.ValidatePrivateKey to confirm that the two values form a valid pair and displays “Keys are valid” or “Keys are invalid” in the main RichTextBox output.

With key management in place, a Transaction class was introduced to encapsulate everything needed for a blockchain transfer. Each transaction records a timestamp, sender address, recipient address, amount and fee. Its constructor concatenates those fields, runs the result through SHA-256 to produce a hash, and then calls Wallet.CreateSignature to sign that hash with the sender’s private key. This signature binds the transaction cryptographically to the sender and prevents tampering.

A Create Transaction button was added to the UI so that, once a valid key pair, recipient address, amount and fee are entered, clicking it constructs a new Transaction and writes its full details into the RichTextBox. This end-to-end flow ensures that every transaction is both authenticated and tamper-evident, laying the groundwork for the transaction pool and balance checks that follow.

In the appendix, the first screenshot will show the application after keys are generated and validated, with “Keys are valid” displayed in the output area. The second screenshot will show a completed transaction in the console, illustrating the timestamp, sender address, recipient address, amount, fee, transaction hash and digital signature.

Part 4

Part four extends the blockchain’s consensus by adding proof-of-work mining, transaction inclusion and miner rewards in the Block class. A new Mine method performs repeated SHA-256 hashing of the block header, which now includes index, timestamp, previous hash, Merkle root and a changing nonce, until the hash meets the difficulty target of four leading zeros. Each time the nonce is incremented, the CreateHash routine combines the header fields and computes the digest. As soon as Mine finds a valid hash, it returns, ensuring every block carries a verifiable proof of work.

When a block is constructed, it now accepts the previous block, the pending transactions list and the miner’s address. A reward transaction is generated that pays a base reward plus any collected fees, and all transactions are fed into a MerkleRoot method to produce the block’s Merkle root. Mine is then invoked to find the nonce and hash. Once mining completes, the block object holds its nonce, hash, Merkle root and the full sorted list of transactions so that every block records both its cryptographic proof and its contained transfers. This is shown in the flowchart before the appendix.

On the blockchain side, the transaction pool was introduced to hold pending transactions, and a GetPendingTransactions method was added to batch them in groups of five. A getLastBlock helper returns the current tip of the chain for linking new blocks. The Windows Forms interface gained three new buttons for Generate New Block, Print All and Print Pending Transactions. Generate New Block retrieves the next batch of transactions, calls the Block constructor to mine and link the new block, adds it to the chain and shows its details in the main output area. Print All lists every block in the chain, and Print Pending Transactions displays any transactions still waiting in the pool.

Part 5

Part five implements a complete validation layer to guarantee the blockchain’s integrity and the correctness of user balances. Two new static methods, ValidateHash and ValidateMerkleRoot, recompute each block’s SHA-256 hash using its index, timestamp, prevHash, nonce and Merkle root and rebuild the Merkle tree over its transactions to ensure nothing has been altered. In the UI, a Full Blockchain Validation button now walks through every block from the second onward, checking that each block’s prevHash matches the actual hash of its predecessor, that its stored hash equals a fresh hash of its header, and that its stored Merkle root matches the recomputed root. If any of these checks fail, the app reports “Blockchain is invalid”; otherwise, it reports “Blockchain is valid.”

To tie balances back to the ledger, a GetBalance method was also added to the Blockchain class. It scans every confirmed transaction in every block, credits amounts where the given address is the recipient and debits the amount plus the fee where it is the sender and returns the net Assignment Coin balance. On the form, you can enter a public key and click Check Balance to display the computed balance in the main console area.

Part 6

In Task 6.1, a multithreaded mining mode was introduced to exploit all available CPU cores and drive down average block-find time. Each thread probes a distinct slice of the nonce space by stepping its nonce counter by the total thread count on every iteration. A shared cancellation flag, guarded by a simple lock, ensures that as soon as one thread finds a valid hash that meets the leading-zero difficulty target, all others stop immediately. A stopwatch measures the elapsed time for each mining run, and those results are written to the console area to compare single-threaded performance with multithreaded runs. The appendix shows comparisons of three multithreaded and three single-threaded blocks and their mining time. The mean block time over the three multithreaded blocks using 8 threads was 6.938 seconds, whereas for single threading, the mean was 39.999 seconds.

Task 6.2 adds an adaptive difficulty algorithm so that block creation stays close to a chosen target interval. After each block is mined, the application checks whether the mining time was faster or slower than the target time. If blocks are being found faster than the target, the difficulty is incremented by one; if too slow, it is decremented, never dropping below one. The current difficulty level is displayed in the UI alongside the mining controls and is logged with every block print-out. As shown in the appendix, as the difficulty increases, so does the mining time until it is longer than the target time, then both the difficulty and mining time decrease again. It is calculated like so:

private int CalculateDifficulty()

{

return prevMiningTime < targetBlockTime ? prevDifficulty + 1 : prevDifficulty - 1;

}

A smoother approach uses an exponential moving average of recent block times. After each block you update the moving average by multiplying the current block time by a weight called alpha and adding the previous moving average multiplied by one minus that weight. You then compute a scaling factor by dividing the moving average by the target block time. The new difficulty is the current difficulty multiplied by the scaling factor rounded to the nearest whole number and never less than one. This makes difficulty adjustments more responsive to sustained changes in computing power while avoiding sudden jumps, helping block creation stay close to the target interval.

Task 6.3 implements transaction selection preferences to explore different miner behaviours. A set of radio buttons in the form lets the user choose between three strategies, including greedy (highest-fee-first), altruistic (oldest-pending-first) or random selection. When “Generate New Block” is clicked, the code sorts the pending-transaction list according to the chosen strategy and then selects the required number of transactions for inclusion in the new block. This makes it easy to observe how each policy affects total fees collected and average transaction wait time. The appendix shows how each strategy chooses 5 transactions from the transaction pool. Once the mode has been determined, it is passed to the GetPendingTransactions function that looks like this:

public List<Transaction> GetPendingTransactions(MiningMode mode)

{

int n = Math.Min(transactionsPerBlock, transactionPool.Count);

if (mode == MiningMode.Greedy)

{

transactionPool.Sort((x, y) => y.fee.CompareTo(x.fee));

}

else if (mode == MiningMode.Altruistic)

{

transactionPool.Sort((x, y) => x.timestamp.CompareTo(y.timestamp));

}

else if (mode == MiningMode.Random)

{

Random rng = new Random();

int i = transactionPool.Count;

while (i > 1)

{

i--;

int k = rng.Next(i + 1);

Transaction value = transactionPool[k];

transactionPool[k] = transactionPool[i];

transactionPool[i] = value;

}

}

List<Transaction> transactions = transactionPool.GetRange(0, n);

transactionPool.RemoveRange(0, n);

return transactions;

}

Part 7

In this final section, a proof-of-stake algorithm was implemented. When new blocks are created through mining, the blocks are “mined”, but when new blocks are created using proof of stake, the blocks are “forged”. Firstly, a “Forge New Block” button was added to the UI, which triggers the proof-of-stake algorithm. This starts by creating a dictionary of address-balance pairs by looking through every block in the chain and every transaction in each block to find all addresses that have either sent or received tokens. Then the getBalance function is called for each address, which populates the stakes dictionary.

These stakes are then used inside a new ValidatorPicker class to call Pick, which shuffles the stakes dictionary, then generates a random value between 0 and the sum of all stakes. It steps through the list, cumulatively adding each address’s stake until the running total meets or exceeds the random sample; at that point, it returns that address as the chosen validator. The mining mode is determined and passed to GetPendingTransactions to add the relevant transactions to the block. This is the ValidatorPicker class:

public static class ValidatorPicker

{

private static readonly Random random = new Random();

public static string Pick(Dictionary<string, double> stakes)

{

if (stakes == null || stakes.Count == 0)

throw new ArgumentException("Stakes dictionary cannot be null or empty.");

double totalStake = stakes.Values.Sum();

if (totalStake <= 0)

throw new InvalidOperationException("Total stake must be positive to select a validator.");

// Shuffle the stakes to remove any order bias

var shuffledStakes = stakes.OrderBy(\_ => random.Next()).ToList();

// Generate a random value in [0, totalStake)

double sample = random.NextDouble() \* totalStake;

double cumulative = 0;

foreach (var kvp in shuffledStakes)

{

cumulative += kvp.Value;

if (cumulative >= sample)

return kvp.Key;

}

return shuffledStakes.Last().Key;

}

}

A new block constructor was created specifically for proof of stake. The timestamp, index, previous block data, transaction list and Merkle Root are all set the same as in the mining constructor. The reward is set to 0.1, and the difficulty is set to 0 because the hash is created instantly without any searching for leading zeros by calling Forge. This is the Forge function:

public String Forge()

{

var stopwatch = Stopwatch.StartNew();

nonce = 0;

String hash = CreateHash(nonce);

stopwatch.Stop();

miningTime = stopwatch.Elapsed.TotalSeconds;

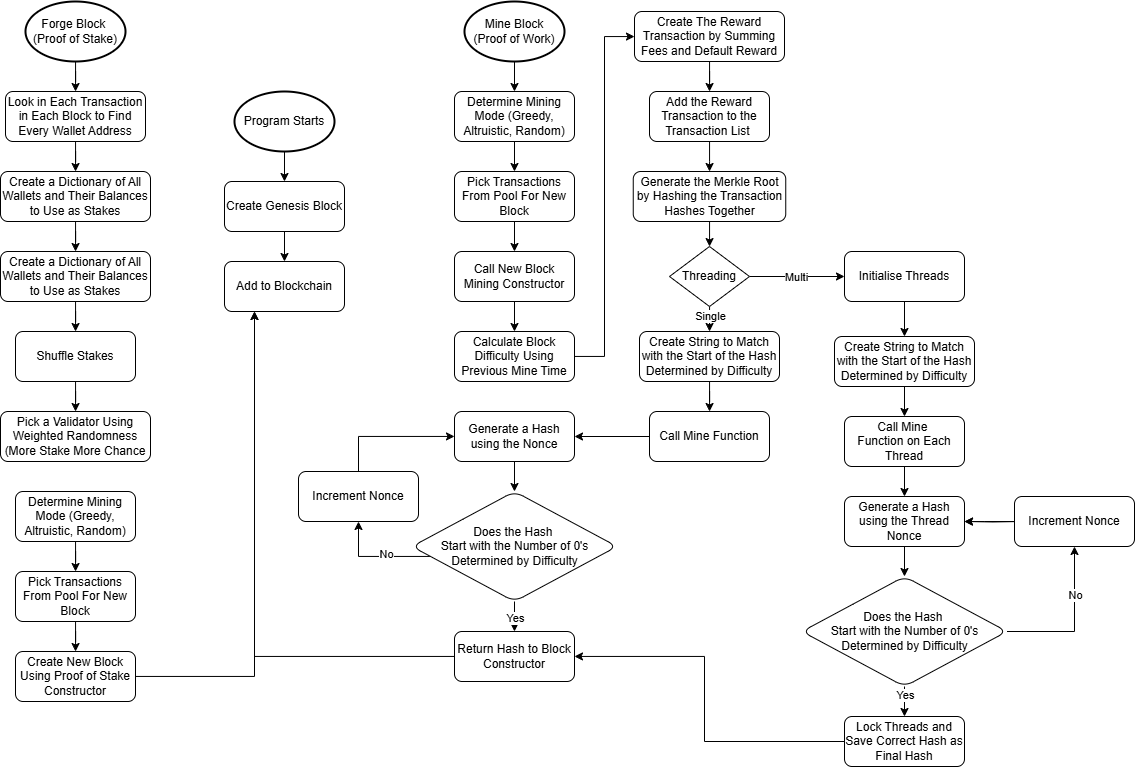
threadingType = "Single (PoS)";

return hash;

}

Once we set the hash, the constructor is finished, so the block is added to the blockchain and the text in the UI is updated to show the blockchain.

Mining and Forging Flow Chart:



Other Button Flow Charts:

A screenshot of a computer

AI-generated content may be incorrect.

Appendix

Part 1

Application window with two textboxes and a button. The text is printed in the large text box from the small text box:

A screenshot of a computer

AI-generated content may be incorrect.

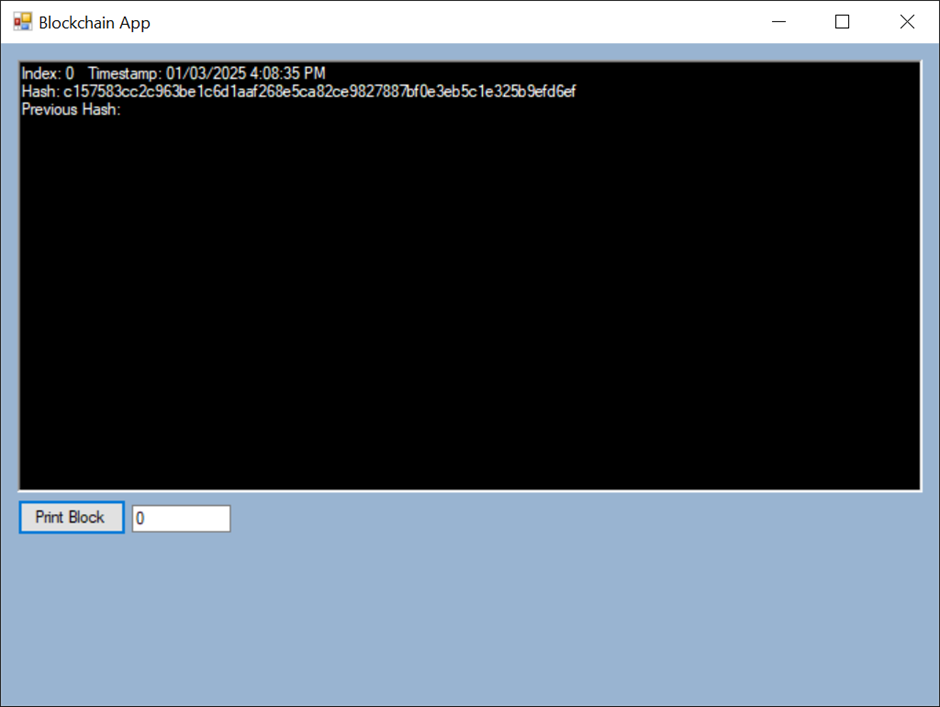
Part 2

New blockchain initialised message when starting the application:

A screenshot of a computer

AI-generated content may be incorrect.

Genesis block:



Part 3

Keys generated in the public and private key text boxes, then validated:

A screenshot of a computer

AI-generated content may be incorrect.

Transactions can be created, but they are not validated on the blockchain:A screenshot of a computer program

AI-generated content may be incorrect.

Part 4

New block and sent a transaction, which is pending:

A screenshot of a computer program

AI-generated content may be incorrect.

Both blocks are shown, with the transaction shown in the second block:A screenshot of a computer

AI-generated content may be incorrect.

Part 5

Keys generated with a transaction generated, giving the wallet tokens:

A screenshot of a computer

AI-generated content may be incorrect.

The transaction in the chain:

A screenshot of a computer

AI-generated content may be incorrect.

Blockchain being validated:

A screenshot of a computer

AI-generated content may be incorrect.

Part 6.1

|  |  |
| --- | --- |
| Threads | Average Block Time |
| 1 | 39.999 |
| 8 | 6.938 |

Multithreaded block 1:

A screenshot of a computer

AI-generated content may be incorrect.

Multithreaded block 2:

A screenshot of a computer

AI-generated content may be incorrect.

Multithreaded block 3:

A screenshot of a computer

AI-generated content may be incorrect.

Single-threaded block 1:

A screenshot of a computer program

AI-generated content may be incorrect.

Single-threaded block 2:

A screenshot of a computer

AI-generated content may be incorrect.

Single-threaded block 3:

A screenshot of a computer

AI-generated content may be incorrect.

Part 6.2

|  |  |  |
| --- | --- | --- |
| Block Index | Difficulty | Mining Time |
| 0 | 4 | 0.211 |
| 1 | 5 | 6.487 |
| 2 | 6 | 24.768 |
| 3 | 5 | 14.148 |

Block 0:

A screenshot of a computer

AI-generated content may be incorrect.

Block 1:

A screenshot of a computer

AI-generated content may be incorrect.

Block 2:

A screenshot of a computer

AI-generated content may be incorrect.

Block 3:

A screenshot of a computer

AI-generated content may be incorrect.

Part 6.3

Three transactions in the pending list with fees from 1 to 3:

A screenshot of a computer

AI-generated content may be incorrect.

Four transactions in the pending list with fees from 4 to 7:

A screenshot of a computer

AI-generated content may be incorrect.

The next block was mined using greedy mining mode, containing the 5 highest fee transactions:

A screenshot of a computer

AI-generated content may be incorrect.

More transactions in the block:

A screenshot of a computer

AI-generated content may be incorrect.

End of block including mining reward transaction:

A screenshot of a computer

AI-generated content may be incorrect.

Using the same 7 transactions as before, we mine a new block using altruistic mode:

A screenshot of a computer

AI-generated content may be incorrect.

Next 3 transactions in the block:

A screenshot of a computer

AI-generated content may be incorrect.

End of block including the mining reward transaction:

A screenshot of a computer

AI-generated content may be incorrect.

Using the same transactions, now we use the random mode for transaction selection and mine a new block:

A screenshot of a computer

AI-generated content may be incorrect.

The next transactions (in random order of fees now):

A screenshot of a computer

AI-generated content may be incorrect.

End of block including mining reward transaction:

A screenshot of a computer

AI-generated content may be incorrect.

Part 7

Forge New Block button added to the UI:

A screenshot of a computer

AI-generated content may be incorrect.

Calculated stake dictionary of address-balance pairs:

A screen shot of a computer screen

AI-generated content may be incorrect.

Random key picked inside ValidatorPicker:

A screen shot of a computer program

AI-generated content may be incorrect.

Hash created in Forge function:

A screen shot of a computer program

AI-generated content may be incorrect.

Forging a new block fails when there are no validators to choose from:

A screenshot of a computer

AI-generated content may be incorrect.

Forged block with 0 difficulty, 0 nonce, near instant mine time and 0.1 reward:

A screen shot of a computer

AI-generated content may be incorrect.

**Code -** [**https://github.com/MrLFW/BlockchainAssignment**](https://github.com/MrLFW/BlockchainAssignment)

// ==== File: Block.cs ====

﻿using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Security.Cryptography;

using System.Text;

using System.Threading;

namespace BlockchainAssignment

{

class Block

{

/\* Block Variables \*/

private DateTime timestamp; // Time of creation

private int index, // Position of the block in the sequence of blocks

difficulty,

prevDifficulty; // An arbitrary number of 0's to proceed a hash value

public String prevHash, // A reference pointer to the previous block

hash, // The current blocks "identity"

merkleRoot, // The merkle root of all transactions in the block

minerAddress; // Public Key (Wallet Address) of the Miner

public List<Transaction> transactionList; // List of transactions in this block

// Proof-of-work

public long nonce; // Number used once for Proof-of-Work and mining

// Rewards

public double reward; // Simple fixed reward established by "Coinbase"

private double targetBlockTime = 15; // Target time between blocks in seconds

private double miningTime;

public double prevMiningTime; // Time taken to mine the previous block

private string threadingType;

const int threadCount = 4; // Hard-coded number of threads

/\* Genesis block constructor \*/

public Block()

{

timestamp = DateTime.Now;

index = 0;

prevMiningTime = 0;

difficulty = 4;

transactionList = new List<Transaction>();

hash = MineMultiThreaded();

}

/\* New Block constructor \*/

public Block(Block lastBlock, List<Transaction> transactions, String minerAddress, bool multithreaded)

{

timestamp = DateTime.Now;

index = lastBlock.index + 1;

prevHash = lastBlock.hash;

prevMiningTime = lastBlock.miningTime;

prevDifficulty = lastBlock.difficulty;

difficulty = CalculateDifficulty();

this.minerAddress = minerAddress; // The wallet to be credited the reward for the mining effort

reward = 1.0; // Assign a simple fixed value reward

transactions.Add(createRewardTransaction(transactions)); // Create and append the reward transaction

transactionList = new List<Transaction>(transactions); // Assign provided transactions to the block

merkleRoot = MerkleRoot(transactionList); // Calculate the merkle root of the blocks transactions

hash = multithreaded ? MineMultiThreaded() : Mine(); // Conduct PoW to create a hash which meets the given difficulty requirement

}

// PROOF OF STAKE CONSTRUCTOR

public Block(Block lastBlock, List<Transaction> transactions, String minerAddress)

{

timestamp = DateTime.Now;

index = lastBlock.index + 1;

prevHash = lastBlock.hash;

prevMiningTime = lastBlock.miningTime;

prevDifficulty = lastBlock.difficulty;

difficulty = 0;

this.minerAddress = minerAddress; // The wallet to be credited the reward for the mining effort

reward = 0.1; // Less reward for proof of stake

transactions.Add(createRewardTransaction(transactions)); // Create and append the reward transaction

transactionList = new List<Transaction>(transactions); // Assign provided transactions to the block

merkleRoot = MerkleRoot(transactionList); // Calculate the merkle root of the blocks transactions

hash = Forge(); // Conduct PoW to create a hash which meets the given difficulty requirement

}

private int CalculateDifficulty()

{

return prevMiningTime < targetBlockTime ? prevDifficulty + 1 : prevDifficulty - 1;

}

/\* Hashes the entire Block object \*/

public String CreateHash(long nonce)

{

String hash = String.Empty;

SHA256 hasher = SHA256Managed.Create();

/\* Concatenate all of the blocks properties including nonce as to generate a new hash on each call \*/

String input = timestamp.ToString() + index + prevHash + nonce + merkleRoot;

/\* Apply the hash function to the block as represented by the string "input" \*/

Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes(input));

/\* Reformat to a string \*/

foreach (byte x in hashByte)

hash += String.Format("{0:x2}", x);

return hash;

}

// Create a Hash which satisfies the difficulty level required for PoW

public String MineMultiThreaded()

{

var stopwatch = Stopwatch.StartNew();

Thread[] miners = new Thread[threadCount];

bool found = false;

String finalHash = String.Empty;

long successfulNonce = 0;

String re = new string('0', difficulty); // A string for analysing the PoW requirement

object lockObject = new object();

void MineThread(object threadId)

{

long threadNonce = (int)threadId; // TODO check we can cast to int then set as long

String hash;

while (!found)

{

hash = CreateHash(threadNonce);

if (hash.StartsWith(re))

{

lock (lockObject)

{

if (!found)

{

found = true;

finalHash = hash;

successfulNonce = threadNonce;

}

}

break;

}

threadNonce += threadCount;

}

}

for (int i = 0; i < threadCount; i++)

{

miners[i] = new Thread(MineThread);

miners[i].Start(i);

}

foreach (var thread in miners)

{

thread.Join();

}

nonce = successfulNonce;

stopwatch.Stop();

miningTime = stopwatch.Elapsed.TotalSeconds;

threadingType = "Multi";

return finalHash;

}

public String Mine()

{

var stopwatch = Stopwatch.StartNew();

nonce = 0; // Initalise the nonce

String hash = CreateHash(nonce); // Hash the block

String re = new string('0', difficulty); // A string for analysing the PoW requirement

while (!hash.StartsWith(re)) // Check the resultant hash against the "re" string

{

nonce++; // Increment the nonce should the difficulty level not be satisfied

hash = CreateHash(nonce); // Rehash with the new nonce as to generate a different hash

}

stopwatch.Stop();

miningTime = stopwatch.Elapsed.TotalSeconds;

threadingType = "Single";

return hash; // Return the hash meeting the difficulty requirement

}

public String Forge()

{

var stopwatch = Stopwatch.StartNew();

nonce = 0;

String hash = CreateHash(nonce);

stopwatch.Stop();

miningTime = stopwatch.Elapsed.TotalSeconds;

threadingType = "Single (PoS)";

return hash;

}

// Merkle Root Algorithm - Encodes transactions within a block into a single hash

public static String MerkleRoot(List<Transaction> transactionList)

{

List<String> hashes = transactionList.Select(t => t.hash).ToList(); // Get a list of transaction hashes for "combining"

// Handle Blocks with...

if (hashes.Count == 0) // No transactions

{

return String.Empty;

}

if (hashes.Count == 1) // One transaction - hash with "self"

{

return HashCode.HashTools.combineHash(hashes[0], hashes[0]);

}

while (hashes.Count != 1) // Multiple transactions - Repeat until tree has been traversed

{

List<String> merkleLeaves = new List<String>(); // Keep track of current "level" of the tree

for (int i = 0; i < hashes.Count; i += 2) // Step over neighbouring pair combining each

{

if (i == hashes.Count - 1)

{

merkleLeaves.Add(HashCode.HashTools.combineHash(hashes[i], hashes[i])); // Handle an odd number of leaves

}

else

{

merkleLeaves.Add(HashCode.HashTools.combineHash(hashes[i], hashes[i + 1])); // Hash neighbours leaves

}

}

hashes = merkleLeaves; // Update the working "layer"

}

return hashes[0]; // Return the root node

}

// Create reward for incentivising the mining of block

public Transaction createRewardTransaction(List<Transaction> transactions)

{

double fees = transactions.Aggregate(0.0, (acc, t) => acc + t.fee); // Sum all transaction fees

return new Transaction("Mine Rewards", minerAddress, (reward + fees), 0, ""); // Issue reward as a transaction in the new block

}

/\* Concatenate all properties to output to the UI \*/

public override string ToString()

{

return "[BLOCK START]"

+ "\nIndex: " + index

+ "\tTimestamp: " + timestamp

+ "\nPrevious Hash: " + prevHash

+ "\n-- PoW --"

+ "\nDifficulty Level: " + difficulty

+ "\nNonce: " + nonce

+ "\nHash: " + hash

+ $"\nMining Time: {miningTime:F3} seconds"

+ "\nThreading Type: " + threadingType

+ "\n-- Rewards --"

+ "\nReward: " + reward

+ "\nMiners Address: " + minerAddress

+ "\n-- " + transactionList.Count + " Transactions --"

+ "\nMerkle Root: " + merkleRoot

+ "\n" + String.Join("\n", transactionList)

+ "\n[BLOCK END]\n";

}

}

}

// ==== File: Blockchain.cs ====

﻿using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BlockchainAssignment

{

class Blockchain

{

// List of block objects forming the blockchain

public List<Block> blocks;

// Maximum number of transactions per block

private int transactionsPerBlock = 5;

// List of pending transactions to be mined

public List<Transaction> transactionPool = new List<Transaction>();

// Default Constructor - initialises the list of blocks and generates the genesis block

public Blockchain()

{

blocks = new List<Block>()

{

new Block() // Create and append the Genesis Block

};

}

// Prints the block at the specified index to the UI

public String GetBlockAsString(int index)

{

// Check if referenced block exists

if (index >= 0 && index < blocks.Count)

return blocks[index].ToString(); // Return block as a string

else

return "No such block exists";

}

// Retrieves the most recently appended block in the blockchain

public Block GetLastBlock()

{

return blocks[blocks.Count - 1];

}

// Retrieve pending transactions and remove from pool

public List<Transaction> GetPendingTransactions(MiningMode mode)

{

int n = Math.Min(transactionsPerBlock, transactionPool.Count);

if (mode == MiningMode.Greedy)

{

transactionPool.Sort((x, y) => y.fee.CompareTo(x.fee));

}

else if (mode == MiningMode.Altruistic)

{

transactionPool.Sort((x, y) => x.timestamp.CompareTo(y.timestamp));

}

else if (mode == MiningMode.Random)

{

Random rng = new Random();

int i = transactionPool.Count;

while (i > 1)

{

i--;

int k = rng.Next(i + 1);

Transaction value = transactionPool[k];

transactionPool[k] = transactionPool[i];

transactionPool[i] = value;

}

}

List<Transaction> transactions = transactionPool.GetRange(0, n);

transactionPool.RemoveRange(0, n);

return transactions;

}

// Check validity of a blocks hash by recomputing the hash and comparing with the mined value

public static bool ValidateHash(Block b)

{

String rehash = b.CreateHash(b.nonce);

return rehash.Equals(b.hash);

}

// Check validity of the merkle root by recalculating the root and comparing with the mined value

public static bool ValidateMerkleRoot(Block b)

{

String reMerkle = Block.MerkleRoot(b.transactionList);

return reMerkle.Equals(b.merkleRoot);

}

// Check the balance associated with a wallet based on the public key

public double GetBalance(String address)

{

// Starting balance

double balance = 100;

// Loop through all approved transactions in order to assess account balance

foreach(Block b in blocks)

{

foreach(Transaction t in b.transactionList)

{

if (t.recipientAddress.Equals(address))

{

balance += t.amount; // Credit funds recieved

}

if (t.senderAddress.Equals(address))

{

balance -= (t.amount + t.fee); // Debit payments placed

}

}

}

return balance;

}

// Output all blocks of the blockchain as a string

public override string ToString()

{

return String.Join("\n", blocks);

}

}

}

// ==== File: BlockchainApp.cs ====

﻿using System;

using System.Collections.Generic;

using System.Linq;

using System.Security.Cryptography.X509Certificates;

using System.Windows.Forms;

namespace BlockchainAssignment

{

public partial class BlockchainApp : Form

{

// Global blockchain object

private Blockchain blockchain;

// Default App Constructor

public BlockchainApp()

{

// Initialise UI Components

InitializeComponent();

// Create a new blockchain

blockchain = new Blockchain();

// Update UI with an initalisation message

UpdateText("New blockchain initialised!");

}

/\* PRINTING \*/

// Helper method to update the UI with a provided message

private void UpdateText(String text)

{

output.Text = text;

}

// Print entire blockchain to UI

private void ReadAll\_Click(object sender, EventArgs e)

{

UpdateText(blockchain.ToString());

}

// Print Block N (based on user input)

private void PrintBlock\_Click(object sender, EventArgs e)

{

if (Int32.TryParse(blockNo.Text, out int index))

UpdateText(blockchain.GetBlockAsString(index));

else

UpdateText("Invalid Block No.");

}

// Print pending transactions from the transaction pool to the UI

private void PrintPendingTransactions\_Click(object sender, EventArgs e)

{

UpdateText(String.Join("\n", blockchain.transactionPool));

}

/\* WALLETS \*/

// Generate a new Wallet and fill the public and private key fields of the UI

private void GenerateWallet()

{

Wallet.Wallet myNewWallet = new Wallet.Wallet(out string privKey);

publicKey.Text = myNewWallet.publicID;

privateKey.Text = privKey;

}

private void GenerateWallet\_Click(object sender, EventArgs e)

{

GenerateWallet();

}

// Validate the keys loaded in the UI by comparing their mathematical relationship

private void ValidateKeys\_Click(object sender, EventArgs e)

{

if (Wallet.Wallet.ValidatePrivateKey(privateKey.Text, publicKey.Text))

UpdateText("Keys are valid");

else

UpdateText("Keys are invalid");

}

// Check the balance of current user

private void CheckBalance\_Click(object sender, EventArgs e)

{

UpdateText(blockchain.GetBalance(publicKey.Text).ToString() + " Assignment Coin");

}

/\* TRANSACTION MANAGEMENT \*/

// Create a new pending transaction and add it to the transaction pool

private void CreateTransaction\_Click(object sender, EventArgs e)

{

Transaction transaction = new Transaction(publicKey.Text, reciever.Text, Double.Parse(amount.Text), Double.Parse(fee.Text), privateKey.Text);

/\* TODO: Validate transaction \*/

blockchain.transactionPool.Add(transaction);

UpdateText(transaction.ToString());

}

private void newBlock(bool multithreaded)

{

var mode = new MiningMode();

if (GreedyButton.Checked)

mode = MiningMode.Greedy;

else if (AltruisticButton.Checked)

mode = MiningMode.Altruistic;

else if (RandomButton.Checked)

mode = MiningMode.Random;

// Retrieve pending transactions to be added to the newly generated Block

List<Transaction> transactions = blockchain.GetPendingTransactions(mode);

// Create and append the new block - requires a reference to the previous block, a set of transactions and the miners public address (For the reward to be issued)

Block newBlock = new Block(blockchain.GetLastBlock(), transactions, publicKey.Text, multithreaded);

blockchain.blocks.Add(newBlock);

UpdateText(blockchain.ToString());

}

/\* BLOCK MANAGEMENT \*/

// Conduct Proof-of-work in order to mine transactions from the pool and submit a new block to the Blockchain

private void newBlockMultiThread\_Click(object sender, EventArgs e)

{

newBlock(multithreaded: true);

}

private void newBlockSingleThread\_Click(object sender, EventArgs e)

{

newBlock(multithreaded: false);

}

/\* BLOCKCHAIN VALIDATION \*/

// Validate the integrity of the state of the Blockchain

private void Validate\_Click(object sender, EventArgs e)

{

// CASE: Genesis Block - Check only hash as no transactions are currently present

if (blockchain.blocks.Count == 1)

{

if (!Blockchain.ValidateHash(blockchain.blocks[0])) // Recompute Hash to check validity

UpdateText("Blockchain is invalid");

else

UpdateText("Blockchain is valid");

return;

}

for (int i = 1; i < blockchain.blocks.Count - 1; i++)

{

if (

blockchain.blocks[i].prevHash != blockchain.blocks[i - 1].hash || // Check hash "chain"

!Blockchain.ValidateHash(blockchain.blocks[i]) || // Check each blocks hash

!Blockchain.ValidateMerkleRoot(blockchain.blocks[i]) // Check transaction integrity using Merkle Root

)

{

UpdateText("Blockchain is invalid");

return;

}

}

UpdateText("Blockchain is valid");

}

private void forgeBlock\_Click(object sender, EventArgs e)

{

var stakes = blockchain.blocks

.SelectMany(b => b.transactionList)

.SelectMany(tx => new[] { tx.senderAddress, tx.recipientAddress })

.Distinct()

.ToDictionary(

addr => addr,

addr => blockchain.GetBalance(addr)

);

if (stakes.Count == 0)

{

UpdateText("Cannot forge new block, no stakes!");

return;

}

var validator = ValidatorPicker.Pick(stakes);

var mode = new MiningMode();

if (GreedyButton.Checked)

mode = MiningMode.Greedy;

else if (AltruisticButton.Checked)

mode = MiningMode.Altruistic;

else if (RandomButton.Checked)

mode = MiningMode.Random;

// Retrieve pending transactions to be added to the newly generated Block

List<Transaction> transactions = blockchain.GetPendingTransactions(mode);

// Create and append the new block - requires a reference to the previous block, a set of transactions and the miners public address (For the reward to be issued)

Block newBlock = new Block(blockchain.GetLastBlock(), transactions, validator);

blockchain.blocks.Add(newBlock);

UpdateText(blockchain.ToString());

}

private void randomTransaction\_Click(object sender, EventArgs e)

{

GenerateRandomTransaction();

}

private void GenerateRandomTransaction()

{

// Generate a random wallet for the sender

GenerateWallet();

string senderAddress = publicKey.Text;

string senderPrivateKey = privateKey.Text;

// Generate a random recipient address

Wallet.Wallet randomWallet = new Wallet.Wallet(out \_);

string recipientAddress = randomWallet.publicID;

// Generate random amount and fee

Random random = new Random();

double amount = Math.Round(random.NextDouble() \* 10, 2); // Random amount between 0 and 10

double fee = Math.Round(random.NextDouble() \* 1, 2); // Random fee between 0 and 1

// Create the transaction

Transaction transaction = new Transaction(senderAddress, recipientAddress, amount, fee, senderPrivateKey);

// Add the transaction to the transaction pool

blockchain.transactionPool.Add(transaction);

// Update the UI with the transaction details

UpdateText(transaction.ToString());

}

}

}

// ==== File: BlockchainApp.Designer.cs ====

﻿namespace BlockchainAssignment

{

partial class BlockchainApp

{

/// <summary>

/// Required designer variable.

/// </summary>

private System.ComponentModel.IContainer components = null;

/// <summary>

/// Clean up any resources being used.

/// </summary>

/// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>

protected override void Dispose(bool disposing)

{

if (disposing && (components != null))

{

components.Dispose();

}

base.Dispose(disposing);

}

#region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

this.output = new System.Windows.Forms.RichTextBox();

this.printBlock = new System.Windows.Forms.Button();

this.blockNo = new System.Windows.Forms.TextBox();

this.generateWallet = new System.Windows.Forms.Button();

this.publicKeyLabel = new System.Windows.Forms.Label();

this.privateKeyLabel = new System.Windows.Forms.Label();

this.publicKey = new System.Windows.Forms.TextBox();

this.privateKey = new System.Windows.Forms.TextBox();

this.validateKeys = new System.Windows.Forms.Button();

this.createTransaction = new System.Windows.Forms.Button();

this.fee = new System.Windows.Forms.TextBox();

this.amount = new System.Windows.Forms.TextBox();

this.feeLabel = new System.Windows.Forms.Label();

this.amountLabel = new System.Windows.Forms.Label();

this.reciever = new System.Windows.Forms.TextBox();

this.recieverKeyLabel = new System.Windows.Forms.Label();

this.newBlockMultiThread = new System.Windows.Forms.Button();

this.printBlockchain = new System.Windows.Forms.Button();

this.readPendingTransactions = new System.Windows.Forms.Button();

this.validate = new System.Windows.Forms.Button();

this.checkBalance = new System.Windows.Forms.Button();

this.currentWalletLabel = new System.Windows.Forms.Label();

this.transactionLabel = new System.Windows.Forms.Label();

this.blocksLabel = new System.Windows.Forms.Label();

this.validationLabel = new System.Windows.Forms.Label();

this.newBlockSingleThread = new System.Windows.Forms.Button();

this.label1 = new System.Windows.Forms.Label();

this.GreedyButton = new System.Windows.Forms.RadioButton();

this.AltruisticButton = new System.Windows.Forms.RadioButton();

this.RandomButton = new System.Windows.Forms.RadioButton();

this.groupBox1 = new System.Windows.Forms.GroupBox();

this.forgeBlock = new System.Windows.Forms.Button();

this.randomTransaction = new System.Windows.Forms.Button();

this.groupBox1.SuspendLayout();

this.SuspendLayout();

//

// output

//

this.output.BackColor = System.Drawing.SystemColors.InfoText;

this.output.ForeColor = System.Drawing.SystemColors.ButtonHighlight;

this.output.Location = new System.Drawing.Point(12, 12);

this.output.Name = "output";

this.output.Size = new System.Drawing.Size(460, 602);

this.output.TabIndex = 0;

this.output.Text = "";

//

// printBlock

//

this.printBlock.Location = new System.Drawing.Point(486, 171);

this.printBlock.Margin = new System.Windows.Forms.Padding(2);

this.printBlock.Name = "printBlock";

this.printBlock.Size = new System.Drawing.Size(66, 25);

this.printBlock.TabIndex = 1;

this.printBlock.Text = "Block #";

this.printBlock.UseVisualStyleBackColor = true;

this.printBlock.Click += new System.EventHandler(this.PrintBlock\_Click);

//

// blockNo

//

this.blockNo.Location = new System.Drawing.Point(556, 174);

this.blockNo.Margin = new System.Windows.Forms.Padding(2);

this.blockNo.Name = "blockNo";

this.blockNo.Size = new System.Drawing.Size(24, 20);

this.blockNo.TabIndex = 2;

//

// generateWallet

//

this.generateWallet.Location = new System.Drawing.Point(11, 660);

this.generateWallet.Margin = new System.Windows.Forms.Padding(2);

this.generateWallet.Name = "generateWallet";

this.generateWallet.Size = new System.Drawing.Size(79, 55);

this.generateWallet.TabIndex = 3;

this.generateWallet.Text = "Generate New Wallet";

this.generateWallet.UseVisualStyleBackColor = true;

this.generateWallet.Click += new System.EventHandler(this.GenerateWallet\_Click);

//

// publicKeyLabel

//

this.publicKeyLabel.AutoSize = true;

this.publicKeyLabel.Location = new System.Drawing.Point(120, 674);

this.publicKeyLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.publicKeyLabel.Name = "publicKeyLabel";

this.publicKeyLabel.Size = new System.Drawing.Size(57, 13);

this.publicKeyLabel.TabIndex = 4;

this.publicKeyLabel.Text = "Public Key";

//

// privateKeyLabel

//

this.privateKeyLabel.AutoSize = true;

this.privateKeyLabel.Location = new System.Drawing.Point(120, 698);

this.privateKeyLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.privateKeyLabel.Name = "privateKeyLabel";

this.privateKeyLabel.Size = new System.Drawing.Size(61, 13);

this.privateKeyLabel.TabIndex = 5;

this.privateKeyLabel.Text = "Private Key";

//

// publicKey

//

this.publicKey.Location = new System.Drawing.Point(185, 671);

this.publicKey.Margin = new System.Windows.Forms.Padding(2);

this.publicKey.Name = "publicKey";

this.publicKey.Size = new System.Drawing.Size(187, 20);

this.publicKey.TabIndex = 6;

//

// privateKey

//

this.privateKey.Location = new System.Drawing.Point(185, 695);

this.privateKey.Margin = new System.Windows.Forms.Padding(2);

this.privateKey.Name = "privateKey";

this.privateKey.Size = new System.Drawing.Size(187, 20);

this.privateKey.TabIndex = 7;

//

// validateKeys

//

this.validateKeys.Location = new System.Drawing.Point(376, 669);

this.validateKeys.Margin = new System.Windows.Forms.Padding(2);

this.validateKeys.Name = "validateKeys";

this.validateKeys.Size = new System.Drawing.Size(87, 22);

this.validateKeys.TabIndex = 8;

this.validateKeys.Text = "Validate Keys";

this.validateKeys.UseVisualStyleBackColor = true;

this.validateKeys.Click += new System.EventHandler(this.ValidateKeys\_Click);

//

// createTransaction

//

this.createTransaction.Location = new System.Drawing.Point(486, 288);

this.createTransaction.Margin = new System.Windows.Forms.Padding(2);

this.createTransaction.Name = "createTransaction";

this.createTransaction.Size = new System.Drawing.Size(196, 26);

this.createTransaction.TabIndex = 9;

this.createTransaction.Text = "Create Transaction";

this.createTransaction.UseVisualStyleBackColor = true;

this.createTransaction.Click += new System.EventHandler(this.CreateTransaction\_Click);

//

// fee

//

this.fee.Location = new System.Drawing.Point(646, 240);

this.fee.Margin = new System.Windows.Forms.Padding(2);

this.fee.Name = "fee";

this.fee.Size = new System.Drawing.Size(36, 20);

this.fee.TabIndex = 13;

//

// amount

//

this.amount.Location = new System.Drawing.Point(577, 240);

this.amount.Margin = new System.Windows.Forms.Padding(2);

this.amount.Name = "amount";

this.amount.Size = new System.Drawing.Size(36, 20);

this.amount.TabIndex = 12;

//

// feeLabel

//

this.feeLabel.AutoSize = true;

this.feeLabel.Location = new System.Drawing.Point(617, 243);

this.feeLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.feeLabel.Name = "feeLabel";

this.feeLabel.Size = new System.Drawing.Size(25, 13);

this.feeLabel.TabIndex = 11;

this.feeLabel.Text = "Fee";

//

// amountLabel

//

this.amountLabel.AutoSize = true;

this.amountLabel.Location = new System.Drawing.Point(530, 243);

this.amountLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.amountLabel.Name = "amountLabel";

this.amountLabel.Size = new System.Drawing.Size(43, 13);

this.amountLabel.TabIndex = 10;

this.amountLabel.Text = "Amount";

//

// reciever

//

this.reciever.Location = new System.Drawing.Point(563, 264);

this.reciever.Margin = new System.Windows.Forms.Padding(2);

this.reciever.Name = "reciever";

this.reciever.Size = new System.Drawing.Size(119, 20);

this.reciever.TabIndex = 15;

//

// recieverKeyLabel

//

this.recieverKeyLabel.AutoSize = true;

this.recieverKeyLabel.Location = new System.Drawing.Point(488, 267);

this.recieverKeyLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.recieverKeyLabel.Name = "recieverKeyLabel";

this.recieverKeyLabel.Size = new System.Drawing.Size(71, 13);

this.recieverKeyLabel.TabIndex = 14;

this.recieverKeyLabel.Text = "Reciever Key";

//

// newBlockMultiThread

//

this.newBlockMultiThread.Location = new System.Drawing.Point(485, 91);

this.newBlockMultiThread.Margin = new System.Windows.Forms.Padding(2);

this.newBlockMultiThread.Name = "newBlockMultiThread";

this.newBlockMultiThread.Size = new System.Drawing.Size(197, 26);

this.newBlockMultiThread.TabIndex = 16;

this.newBlockMultiThread.Text = "Generate New Block (Multi Thread)";

this.newBlockMultiThread.UseVisualStyleBackColor = true;

this.newBlockMultiThread.Click += new System.EventHandler(this.newBlockMultiThread\_Click);

//

// printBlockchain

//

this.printBlockchain.Location = new System.Drawing.Point(584, 171);

this.printBlockchain.Margin = new System.Windows.Forms.Padding(2);

this.printBlockchain.Name = "printBlockchain";

this.printBlockchain.Size = new System.Drawing.Size(98, 25);

this.printBlockchain.TabIndex = 17;

this.printBlockchain.Text = "Entire Chain";

this.printBlockchain.UseVisualStyleBackColor = true;

this.printBlockchain.Click += new System.EventHandler(this.ReadAll\_Click);

//

// readPendingTransactions

//

this.readPendingTransactions.Location = new System.Drawing.Point(486, 318);

this.readPendingTransactions.Margin = new System.Windows.Forms.Padding(2);

this.readPendingTransactions.Name = "readPendingTransactions";

this.readPendingTransactions.Size = new System.Drawing.Size(196, 25);

this.readPendingTransactions.TabIndex = 18;

this.readPendingTransactions.Text = "Read Pending Transactions";

this.readPendingTransactions.UseVisualStyleBackColor = true;

this.readPendingTransactions.Click += new System.EventHandler(this.PrintPendingTransactions\_Click);

//

// validate

//

this.validate.Location = new System.Drawing.Point(486, 669);

this.validate.Margin = new System.Windows.Forms.Padding(2);

this.validate.Name = "validate";

this.validate.Size = new System.Drawing.Size(196, 46);

this.validate.TabIndex = 19;

this.validate.Text = "Full Blockchain Validation";

this.validate.UseVisualStyleBackColor = true;

this.validate.Click += new System.EventHandler(this.Validate\_Click);

//

// checkBalance

//

this.checkBalance.Location = new System.Drawing.Point(376, 692);

this.checkBalance.Margin = new System.Windows.Forms.Padding(2);

this.checkBalance.Name = "checkBalance";

this.checkBalance.Size = new System.Drawing.Size(87, 24);

this.checkBalance.TabIndex = 20;

this.checkBalance.Text = "Check Balance";

this.checkBalance.UseVisualStyleBackColor = true;

this.checkBalance.Click += new System.EventHandler(this.CheckBalance\_Click);

//

// currentWalletLabel

//

this.currentWalletLabel.AutoSize = true;

this.currentWalletLabel.Location = new System.Drawing.Point(11, 645);

this.currentWalletLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.currentWalletLabel.Name = "currentWalletLabel";

this.currentWalletLabel.Size = new System.Drawing.Size(74, 13);

this.currentWalletLabel.TabIndex = 22;

this.currentWalletLabel.Text = "Current Wallet";

//

// transactionLabel

//

this.transactionLabel.AutoSize = true;

this.transactionLabel.Location = new System.Drawing.Point(483, 216);

this.transactionLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.transactionLabel.Name = "transactionLabel";

this.transactionLabel.Size = new System.Drawing.Size(68, 13);

this.transactionLabel.TabIndex = 23;

this.transactionLabel.Text = "Transactions";

//

// blocksLabel

//

this.blocksLabel.AutoSize = true;

this.blocksLabel.Location = new System.Drawing.Point(483, 46);

this.blocksLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.blocksLabel.Name = "blocksLabel";

this.blocksLabel.Size = new System.Drawing.Size(39, 13);

this.blocksLabel.TabIndex = 24;

this.blocksLabel.Text = "Blocks";

//

// validationLabel

//

this.validationLabel.AutoSize = true;

this.validationLabel.Location = new System.Drawing.Point(489, 654);

this.validationLabel.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.validationLabel.Name = "validationLabel";

this.validationLabel.Size = new System.Drawing.Size(53, 13);

this.validationLabel.TabIndex = 25;

this.validationLabel.Text = "Validation";

//

// newBlockSingleThread

//

this.newBlockSingleThread.Location = new System.Drawing.Point(485, 61);

this.newBlockSingleThread.Margin = new System.Windows.Forms.Padding(2);

this.newBlockSingleThread.Name = "newBlockSingleThread";

this.newBlockSingleThread.Size = new System.Drawing.Size(197, 26);

this.newBlockSingleThread.TabIndex = 26;

this.newBlockSingleThread.Text = "Generate New Block (Single Thread)";

this.newBlockSingleThread.UseVisualStyleBackColor = true;

this.newBlockSingleThread.Click += new System.EventHandler(this.newBlockSingleThread\_Click);

//

// label1

//

this.label1.AutoSize = true;

this.label1.Location = new System.Drawing.Point(482, 383);

this.label1.Margin = new System.Windows.Forms.Padding(2, 0, 2, 0);

this.label1.Name = "label1";

this.label1.Size = new System.Drawing.Size(0, 13);

this.label1.TabIndex = 27;

//

// GreedyButton

//

this.GreedyButton.AutoSize = true;

this.GreedyButton.Checked = true;

this.GreedyButton.Location = new System.Drawing.Point(31, 19);

this.GreedyButton.Name = "GreedyButton";

this.GreedyButton.Size = new System.Drawing.Size(59, 17);

this.GreedyButton.TabIndex = 28;

this.GreedyButton.TabStop = true;

this.GreedyButton.Text = "Greedy";

this.GreedyButton.UseVisualStyleBackColor = true;

//

// AltruisticButton

//

this.AltruisticButton.AutoSize = true;

this.AltruisticButton.Location = new System.Drawing.Point(31, 42);

this.AltruisticButton.Name = "AltruisticButton";

this.AltruisticButton.Size = new System.Drawing.Size(64, 17);

this.AltruisticButton.TabIndex = 29;

this.AltruisticButton.TabStop = true;

this.AltruisticButton.Text = "Altruistic";

this.AltruisticButton.UseVisualStyleBackColor = true;

//

// RandomButton

//

this.RandomButton.AutoSize = true;

this.RandomButton.Location = new System.Drawing.Point(31, 65);

this.RandomButton.Name = "RandomButton";

this.RandomButton.Size = new System.Drawing.Size(65, 17);

this.RandomButton.TabIndex = 30;

this.RandomButton.TabStop = true;

this.RandomButton.Text = "Random";

this.RandomButton.UseVisualStyleBackColor = true;

//

// groupBox1

//

this.groupBox1.Controls.Add(this.GreedyButton);

this.groupBox1.Controls.Add(this.AltruisticButton);

this.groupBox1.Controls.Add(this.RandomButton);

this.groupBox1.Location = new System.Drawing.Point(485, 406);

this.groupBox1.Name = "groupBox1";

this.groupBox1.Size = new System.Drawing.Size(200, 94);

this.groupBox1.TabIndex = 32;

this.groupBox1.TabStop = false;

this.groupBox1.Text = "Mining Mode";

//

// forgeBlock

//

this.forgeBlock.Location = new System.Drawing.Point(485, 131);

this.forgeBlock.Margin = new System.Windows.Forms.Padding(2);

this.forgeBlock.Name = "forgeBlock";

this.forgeBlock.Size = new System.Drawing.Size(197, 26);

this.forgeBlock.TabIndex = 33;

this.forgeBlock.Text = "Forge New Block";

this.forgeBlock.UseVisualStyleBackColor = true;

this.forgeBlock.Click += new System.EventHandler(this.forgeBlock\_Click);

//

// randomTransaction

//

this.randomTransaction.Location = new System.Drawing.Point(486, 347);

this.randomTransaction.Margin = new System.Windows.Forms.Padding(2);

this.randomTransaction.Name = "randomTransaction";

this.randomTransaction.Size = new System.Drawing.Size(196, 26);

this.randomTransaction.TabIndex = 34;

this.randomTransaction.Text = "Generate Random Transaction";

this.randomTransaction.UseVisualStyleBackColor = true;

this.randomTransaction.Click += new System.EventHandler(this.randomTransaction\_Click);

//

// BlockchainApp

//

this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);

this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

this.BackColor = System.Drawing.SystemColors.ActiveCaption;

this.ClientSize = new System.Drawing.Size(697, 726);

this.Controls.Add(this.randomTransaction);

this.Controls.Add(this.forgeBlock);

this.Controls.Add(this.groupBox1);

this.Controls.Add(this.label1);

this.Controls.Add(this.createTransaction);

this.Controls.Add(this.newBlockSingleThread);

this.Controls.Add(this.validationLabel);

this.Controls.Add(this.blocksLabel);

this.Controls.Add(this.transactionLabel);

this.Controls.Add(this.currentWalletLabel);

this.Controls.Add(this.checkBalance);

this.Controls.Add(this.validate);

this.Controls.Add(this.readPendingTransactions);

this.Controls.Add(this.printBlockchain);

this.Controls.Add(this.newBlockMultiThread);

this.Controls.Add(this.reciever);

this.Controls.Add(this.recieverKeyLabel);

this.Controls.Add(this.fee);

this.Controls.Add(this.amount);

this.Controls.Add(this.feeLabel);

this.Controls.Add(this.amountLabel);

this.Controls.Add(this.validateKeys);

this.Controls.Add(this.privateKey);

this.Controls.Add(this.publicKey);

this.Controls.Add(this.privateKeyLabel);

this.Controls.Add(this.publicKeyLabel);

this.Controls.Add(this.generateWallet);

this.Controls.Add(this.blockNo);

this.Controls.Add(this.printBlock);

this.Controls.Add(this.output);

this.ForeColor = System.Drawing.Color.Black;

this.Name = "BlockchainApp";

this.Text = "Blockchain App";

this.groupBox1.ResumeLayout(false);

this.groupBox1.PerformLayout();

this.ResumeLayout(false);

this.PerformLayout();

}

#endregion

private System.Windows.Forms.RichTextBox output;

private System.Windows.Forms.Button printBlock;

private System.Windows.Forms.TextBox blockNo;

private System.Windows.Forms.Button generateWallet;

private System.Windows.Forms.Label publicKeyLabel;

private System.Windows.Forms.Label privateKeyLabel;

private System.Windows.Forms.TextBox publicKey;

private System.Windows.Forms.TextBox privateKey;

private System.Windows.Forms.Button validateKeys;

private System.Windows.Forms.Button createTransaction;

private System.Windows.Forms.TextBox fee;

private System.Windows.Forms.TextBox amount;

private System.Windows.Forms.Label feeLabel;

private System.Windows.Forms.Label amountLabel;

private System.Windows.Forms.TextBox reciever;

private System.Windows.Forms.Label recieverKeyLabel;

private System.Windows.Forms.Button newBlockMultiThread;

private System.Windows.Forms.Button printBlockchain;

private System.Windows.Forms.Button readPendingTransactions;

private System.Windows.Forms.Button validate;

private System.Windows.Forms.Button checkBalance;

private System.Windows.Forms.Label currentWalletLabel;

private System.Windows.Forms.Label transactionLabel;

private System.Windows.Forms.Label blocksLabel;

private System.Windows.Forms.Label validationLabel;

private System.Windows.Forms.Button newBlockSingleThread;

private System.Windows.Forms.Label label1;

private System.Windows.Forms.RadioButton GreedyButton;

private System.Windows.Forms.RadioButton AltruisticButton;

private System.Windows.Forms.RadioButton RandomButton;

private System.Windows.Forms.GroupBox groupBox1;

private System.Windows.Forms.Button forgeBlock;

private System.Windows.Forms.Button randomTransaction;

}

}

// ==== File: MiningMode.cs ====

﻿using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BlockchainAssignment

{

public enum MiningMode

{

Greedy,

Altruistic,

Random,

}

}

// ==== File: Program.cs ====

﻿using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace BlockchainAssignment

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new BlockchainApp());

}

}

}

// ==== File: StakeValidator.cs ====

using System;

using System.Collections.Generic;

using System.Security.Cryptography;

using System.Text;

namespace BlockchainAssignment

{

public static class StakeValidator

{

// Minimum balance required to participate in staking

private const double MINIMUM\_STAKE = 1.0;

// Time in seconds between stakes

private const int STAKE\_TIME\_WINDOW = 60;

// Factor to weight stake amount in validation

private const double STAKE\_WEIGHT\_FACTOR = 0.1;

// Verify if an address is eligible to create the next block based on their stake

//public static bool ValidateStake(string address, double balance, DateTime lastStakeTime, Block block)

//{

// // Check if validator has minimum required stake

// if (balance < MINIMUM\_STAKE)

// return false;

// // Calculate time since last stake

// TimeSpan coinAge = DateTime.Now - lastStakeTime;

// // Calculate stake weight

// double stakeWeight = CalculateStakeWeight(balance, coinAge);

// // Generate stake hash

// using (var sha256 = SHA256.Create())

// {

// var input = Encoding.UTF8.GetBytes(address + lastStakeTime.Ticks + stakeWeight);

// var hash = sha256.ComputeHash(input);

// // Convert first 8 bytes to value between 0 and 1

// ulong value = BitConverter.ToUInt64(hash, 0);

// double normalized = value / (double)ulong.MaxValue;

// // Higher stake weight = higher chance of being selected

// return normalized < (stakeWeight / 100.0);

// }

//}

}

}

// ==== File: Transaction.cs ====

﻿using System;

using System.Collections.Generic;

using System.Linq;

using System.Security.Cryptography;

using System.Text;

using System.Threading.Tasks;

namespace BlockchainAssignment

{

class Transaction

{

/\* Transaction Variables \*/

public DateTime timestamp; // Time of creation

public String senderAddress, recipientAddress; // Participants public key addresses

public double amount, fee; // Quantities transferred

public String hash, signature; // Attributes for verification of validity

/\* Transaction Constructor \*/

public Transaction(String from, String to, double amount, double fee, String privateKey)

{

timestamp = DateTime.Now;

senderAddress = from;

recipientAddress = to;

this.amount = amount;

this.fee = fee;

hash = CreateHash(); // Hash the transaction attributes

signature = Wallet.Wallet.CreateSignature(from, privateKey, hash); // Sign the hash with the senders private key ensuring validity

}

/\* Hash the transaction attributes using SHA256 \*/

public String CreateHash()

{

String hash = String.Empty;

SHA256 hasher = SHA256Managed.Create();

/\* Concatenate all transaction properties \*/

String input = timestamp + senderAddress + recipientAddress + amount + fee;

/\* Apply the hash function to the "input" string \*/

Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes(input));

/\* Reformat to a string \*/

foreach (byte x in hashByte)

hash += String.Format("{0:x2}", x);

return hash;

}

// Represent a transaction as a string for output to UI

public override string ToString()

{

return "\n [TRANSACTION START]"

+ "\n Timestamp: " + timestamp

+ "\n -- Verification --"

+ "\n Hash: " + hash

+ "\n Signature: " + signature

+ "\n -- Quantities --"

+ "\n Transferred: " + amount + " Assignment Coin"

+ "\t Fee: " + fee

+ "\n -- Participants --"

+ "\n Sender: " + senderAddress

+ "\n Reciever: " + recipientAddress

+ "\n [TRANSACTION END]\n";

}

}

}

// ==== File: ValidatorPicker.cs ====

﻿using System;

using System.Collections.Generic;

using System.Linq;

namespace BlockchainAssignment

{

public static class ValidatorPicker

{

private static readonly Random random = new Random();

public static string Pick(Dictionary<string, double> stakes)

{

if (stakes == null || stakes.Count == 0)

throw new ArgumentException("Stakes dictionary cannot be null or empty.");

double totalStake = stakes.Values.Sum();

if (totalStake <= 0)

throw new InvalidOperationException("Total stake must be positive to select a validator.");

// Shuffle the stakes to remove any order bias

var shuffledStakes = stakes.OrderBy(\_ => random.Next()).ToList();

// Generate a random value in [0, totalStake)

double sample = random.NextDouble() \* totalStake;

double cumulative = 0;

foreach (var kvp in shuffledStakes)

{

cumulative += kvp.Value;

if (cumulative >= sample)

return kvp.Key;

}

return shuffledStakes.Last().Key;

}

}

}