**Module Code:** CS3BC20

**Assignment report Title:** Blockchain Coursework Assignment

**Student Number: 26019697**

**Date: 17/03/2021**

**Actual hrs spent for the assignment: 25**

**P1 - Project Setup**

To setup the project, the provided files were imported to Visual Studio code as a solution. The project actually (surprisingly) worked straight out of the box.

Visual Studio was already set up as required as it had been used before. After a brief look at the structure of the project I was ready to start on the first actual task.

**P2 - Blocks and the Blockchain**

First of all the classes Block.cs and Blockchain.cs were created, the Blockchain.cs included a list of blocks named Blocks. An instance of the Blockchain is instantiated in the constructor for BlockchainApp.cs, it was followed by an update to the main text box saying "Blockchain Initialised!".

Next the attributes of the Block must be added - for now simple information on the Block is used (more will be added later). The block now contains a timestamp (block creation time), an index for its position in the blockchain (every time a new block is added, the index of the block will increase). Hash of the block - this will hold the SHA256 hash of all the data stored inside of the block - and hash of the previous block - this holds the hash of the previous block and will be used to verify the blockchain later on - it is also used to create the hash of the current block.

After these 4 basic attributes have been added work can be done on initialising the values of the attributes through a constructor. We will need two types of constructor for this class. The first type is for the genesis block and will take no arguments, the second type is for all the other blocks and will use data from the previous block to generate data for the current block. The genesis block constructor will be called on initialisation of the application, where it is added to the blockchain in Blockchain.cs. For now, the new block constructor is passed the previous block, from this the previous hash and block index can be extracted by setting:

index = prevBlock.index + 1;

and

prevHash = prevBlock.hash;

Now there is some basic information for the block class, a function for generating a hash from this block is needed, so each block has its own hash and can pass it to the next block, and hence create a blockchain. To generate a hash, SHA256 is used, this takes any length of values and converts it into a fixed length hexadecimal value (values from 0-F). All the data from the block is concatenated in a function called createHash(), where it is then hashed using SHA256. For now, a simple generated hash is all we need to generate blocks, so we store the value from createHash() in the hash value for the block.

A button is added to the UI with a text box, it takes the value from the text box as an index, and gets the block information for that block and shows it in the main text box using

richTextBox1.Text = blockchain.getBlock(index.Text).getData()

At the end of this section there is simple Block with simple data (time, index, previousHash and hash). There is a constructor for the genesis block. There is also a way of generating a new block, and generating a hash for the new block based off of all the information in the block. A Blockchain is instantiated in the BlockchainApp.cs upon initialisation of the block, which immediately calls the constructor for the genesis block creating the first block, allowing more blocks to be created. There is no point in creating blocks yet as they have no information to store - the next section starts to create the information to be stored in the blocks.

**P3 - Transactions and Digital Signatures**

Next the data for the blocks (transactions) and information about these transactions must be generated, now useful blocks can be added to the blockchain.

Elliptic Curve Digital Signature Algorithm (ECDSA) is an asymmetrical encryption algorithm which is used for "signing" the blocks. It is done by encrypting the transaction data with the senders private key, this signature can then be reversed using the senders private key in order to verify the sender is who they say they are.

Most of the methods of the encryption have already been provided, however a brief description of the included functions is included.

Wallet(out String privateKey) - this is the constructor for the wallet, it generates a private and public key pair for the wallet.

ValidatePrivateKey(privateKey, publicID) - this checks that the public and private key are correctly paired. Returns True if they are mathematically related (a true asymmetric key pair).

CreateSignature(publicID, privateKey, data) - this function takes the data and signs it using the private key and public id. If the key creates is null then returns null.

ValidateSignature(publicID, data, digitalSig) - this function validates that the data has been signed with the correct private key that is related to the public key.

To implement transactions a wallet must first be generated so a button which generates a wallet is created. This wallet shows the public and private key in texts boxes on the screen. This allows us to get the public/private keys for transactions as they are stored on screen. Additionally, a button to validate the privatekey is added, which calls ValidatePrivateKey(privateKey, publicID) to check that the public/private key pair are legitimate. If legitimate "Keys validated successfully" is printed in the richTextBox.

Creating a Transaction class is now required that transaction data can be stored in a single object which is what will later form part of the blocks in the blockchain. The following data is added to the transaction class: hash (created using generateHash()), signature (created using public, private keys and the data), sender address, recipient address, time of transaction, amount sent and fee for sending the transaction. Sender address, recipient address, amount and fee are all provided from data on the UI, and timestamp, hash and signature are all generated automatically in the constructor - in that order, as each is required for the next.

The private key is not saved anywhere as a variable, and is passed from the Wallet instance (which would be local in a true P2P network - slightly different in ours as it is not online) to sign the transaction data.

Now data from the UI can be input to create a transaction, somewhere is needed to store the pending transactions for them to be added to the blocks when they are generated. To store the pending transactions a List of transactions called pendingTransactions is created in the Blockchain class. This stores all the pending transactions, so they can be added to the blocks when they are implemented in the future.

Pending transactions are not verified until they have been added to a block on the blockchain and hence verified by miners. (There will be more checks such as does the sender have enough balance, which will be added later on.) Unverified or pending transactions are stored in the Blockchain class as pending transactions until they are added to blocks - this will be explained in the next section.

**P4 - Consensus Algorithms (Proof Of Work)**

Now there are pending transactions, but no way for them to be added to the blockchain. To do this a way to generate blocks which contain some of the transactions from the pending transactions list must be implemented.

To add transactions to a block from the pending transaction list created previously, a list of the 10 most recent transactions are added to the constructor of the new block, this constructed object is immediately added to the block chain like:

blockchain.addBlock(new Block(blockchain.getBlockchainSize(), blockchain.getLastBlock().getHash(), blockchain.pendingTransactions, publicKey.Text));

The block constructor uses getBlockchainSize() to get the index of the new block, getLastBlock() is used to get parts of the the old block that are required such as difficulty and the hash for the previous block, the blockchain.pendingTransactions is passed so the most recent 10 transactions can be added to the blockchain, and the publicKey.text() is sent for mining rewards (discussed later).

There needs to be a way to see created blocks, so a button on the UI is created which will get all the blocks, perform getData() on each of them, and then print the result to the screen - allowing all the blocks in the blockchain to be viewed.

To add to the method of getting and displaying all of the block information, the getData() function for the block object now also returns all the data about each of the transactions held within the block. This means it is printed to the screen when the Show All Blocks button is pressed.

Proof of work is implemented next, this is done by hashing all the data in the block - this includes the nonce, which will be incremented if the difficulty level is not met. The difficulty level is set to 5, this means that if a hash does not start with 5 0's then the hash is rejected, the nonce is incremented (to generate a different hash) and then all the data is rehashed. This process is repeated until the hash result has 5 leading zeros, once it does then the difficulty measure is considered met and the node which achieved the hash with the 5 leading zeros is awarded a mining reward. The incremented value of the nonce is stored, along with the correct hash, once added to the blockchain these values are immutable.

Now there is proof of work implemented, with a difficulty level making it harder to solve the SHA256 hash, an incentive must be provided to miners to convince them to mine, these are called block rewards. There are two types of block reward, a set reward which is used to distribute the currency amongst the community and fees which are what users pay to send money. Both of these rewards are given to the miner who first reaches the solution of the correct difficulty. The rewards are sent to the miners public address and are send from the MiningRewards address.

The rewards are added as another transaction on the end of each block, and it is added as a parameter to the createHash() method as another transaction in the transaction list in the block. Now the blockchain is fully working with difficulty added for the miners, and the blocks with the transactions being able to be added to the blockchain.

**P5 - Validation**

Now there is a working blockchain, validation must be added to confirm that everything is added correctly. All the blocks can be added using information from the previous block, and this is how it can be verified that each of the blocks are correct.

The first check that is implemented is that the balance of the wallet is correct. This is done by checking all the blocks in the blockchain and any blocks which contain the address of the wallet record whether money is added to or removed from the wallet. This can be used to verify the balance by checking that the current balance is what is should be in regards to all of the previous blocks and their transactions.

Additionally validation of the blocks and the Merkle root needs to be performed to ensure that none of the transactions have been tampered with. This was done by using a passing a list of all the hashes to a recursive function which created a tree from the list of hashes and then gradually reduces the tree with each function call, combining hashes in order to generate the Merkle root. The Merkle root is then used for each new block that is added to the blockchain - this provides integrity and ensures that any tampering will be noticed.

If the Merkle root is different from the Merkle root calculated from all the previous blocks, then it can be discerned that there has been some tampering along the way.

Now all of the methods of validation have been discussed, they can all be implemented into a single function which is available by a button on the UI. The function does the following:

* Checks the Merkle root is valid by recalculating the Merkle root.
* Checks the hashes are all correct by ensuring that block[i].prevHash = block[i-1].hash
* Checks the signature is correct for all transactions sent and validates the balance in the wallet

**P6 - Coursework Tasks**

**Task 1 - Threading during mining.**

To speed up the process of mining increasing the number of cores aka threading can be used. To implement threading I simply created threads which do the following:

* Get the nonce value from the block class and increment the nonce value. This is done in a lock to avoid multiple reads/writes to the main program nonce value.

lock (locker)

{

myNonce = this.nonce++;

}

* Note: There are potentially better ways to allocate the nonce value which require no locking and hence no waiting i.e. round-robin, however I wanted to test my understanding of using locks.
* Then outside the lock in the new thread, the computation is performed using the newly acquired myNonce, the difficulty check is also performed.
* If the result reaches the difficulty level then the global variable done is set to true to stop the other threads, the nonce of the class is set to be equal to the nonce of the thread, and the hash of the class is set to be equal to the one of the thread. This is all performed within a lock as they are shared variables.

lock (locker)

{

if (difficultyCheck == difficultyString && !done)

{

done = true

// set the nonce equal to the nonce used in this thread

this.nonce = myNonce;

this.hash = hash;

}

}

This is a pretty simple example of threading to speed up the computation, but as observable can see below there are definitely benefits to running the process on multiple cores. It overcomes the double work problem by incrementing the nonce whilst locked, this means that each thread can only read a distinct value of the nonce.

Timing the mining process (with difficultly level 5). Results are in seconds. Each test run 20 times

|  |  |  |
| --- | --- | --- |
| Single Thread | Multiple Threads (4) | Multiple Threads (6) |
| 4.84 | 1.24 | 0.12 |
| 14.98 | 1.80 | 4.27 |
| 16.18 | 2.12 | 5.18 |
| 13.81 | 5.06 | 5.86 |
| 7.87 | 2.60 | 2.12 |
| 11.30 | 1.59 | 3.01 |
| 26.23 | 2.24 | 0.92 |
| 9.83 | 5.48 | 2.44 |
| 9.44 | 2.32 | 0.36 |
| 17.47 | 0.59 | 2.69 |
| 6.07 | 21.00 | 2.65 |
| 11.61 | 2.26 | 3.86 |
| 25.96 | 14.46 | 1.12 |
| 5.70 | 0.25 | 1.24 |
| 0.16 | 2.86 | 3.61 |
| 6.48 | 2.50 | 1.43 |
| 1.72 | 1.46 | 2.17 |
| 15.32 | 0.81 | 3.95 |
| 12.09 | 1.22 | 7.72 |
| 27.51 | 13.24 | 5.52 |
| Avg: 12.2285 | Avg 4.255 | Avg 3.012 |

Speedup (4): 2.87

Speedup (6): 4.06

When are using 4 threads we are expecting around 4x speedup - but we get slightly less due to overheads and resource locking (and small computation time). Results above show just under 3x speed up, this could be due to small sample size with three values over 10 seconds - the other sample had no values over 30 seconds (assuming ~3x speedup), implying that there were more computations (hashes) done by the threaded application before.

As I have a 6 core machine I also tried to run it with 6 threads - which achieved a speedup of around x4. This is a good amount of speedup and shows the threaded application clearly runs significantly faster than the regular one.

**Notes:** After this task was performed, I spotted an optimisation in my code that could be made. The lines:

String input = index.ToString() + time.ToString() + prevHash + nonce + difficulty + reward + merkleRoot;

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

Were combined to make:

Byte[] hashByte = hasher.ComputeHash(Encoding.UTF8.GetBytes((index.ToString() + time.ToString() + prevHash + nonce + difficulty + reward + merkleRoot)));

This avoids writing to a variable each time a hash is needed - which should increase the speed of the code.

**Task 2 - Mining Settings**

Potential use case for each preference

When transactions are picked from the pool at the moment, they are picked in a FIFO (first in first out) order - also known as altruistic, this is one way of picking the transactions from the transaction pool. In this task 3 new ways are implemented to pick values from the pending transactions.

Greedy - this method chooses the transactions with the highest fees, as fees are decided by the sender, if a sender increases the fees then the transaction will process faster. This is an interesting method as it allow people using the blockchain to decide on how fast they want their transactions to go through. This does have the drawback that transactions with very low fees make never process or take an extended period of time to process. <https://eprint.iacr.org/2013/881.pdf>

Altruistic - this method also known as first in first out is the current method used in Bitcoin, and is very simple, the first transactions submitted are the first transactions added to blocks. No change was required for implementation of this method. <https://www.coindesk.com/bitcoin-paves-a-way-for-evolution-of-the-species>

Address Preference - this method chooses transactions based on a user provided preference, this allows a user to mine their own transactions quickly (provided they can beat other miners). If the transactions from the specified address are less than the number of transactions per block or are not available at all, the method defaults to altruistic selection.

Random - this method randomly chooses transactions from the transaction list, adding them to the block and then removing them from the main pendingTransaction list.

Machine generated alternative text:
Blockchain App 
New blockchah iråiabsed! 
Pnnt Block 
New Block 
Create Wallet 
Validate Keys 
Show Al 
Penhg 
x 
Make 
Transactio 
Validate 
Block chain 
Anomt 
Fee 
Balance 
Trænsaction Pick 

**Fig 1. Final screen of the blockchain application.**