

ECASH.JS
A PEER-TO-PEER ELECTRONIC CASH SYSTEM

A Project

Presented to the faculty of the Department of Computer Science
California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Rijul Luman

SPRING
2019

© 2019

Rijul Luman

ALL RIGHTS RESERVED

ii

ECASH.JS
A PEER-TO-PEER ELECTRONIC CASH SYSTEM

A Project

by

Rijul Luman

Approved by:

_____, Committee Chair
Dr. Ted Krovetz

_____, Second Reader
Dr. Haiquan Chen

Date

Student: Rijul Luman

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

_____, Graduate Coordinator
Dr. Jinsong Ouyang

Date

Department of Computer Science

Abstract
of
ECASH.JS
A PEER-TO-PEER ELECTRONIC CASH SYSTEM
by
Rijul Luman

Since the rise of internet and e-commerce, buying and selling goods over the internet has relied heavily on financial institutions acting as 3rd parties to process financial transactions. These 3rd parties often charge a good percentage of the payment as transaction fees and often take days to complete. These processes are therefore based upon the 2 parties trusting a 3rd party to process their transaction, and as a 3rd party must be “trusted” there is always room for a transaction to be reversed. Before Bitcoin, there was no way to make a non-reversible payment online for a non-reversible service as there is with cash in the physical world. With the recent price rise of Bitcoin, we have witnessed the impact a decentralized digital currency can have on the world. Mining Bitcoins require special hardware; thus, I have developed a coin using JavaScript, which can be mined on any device with minimal overhead. All services are accessible via REST APIs to the Full Node.

This project contains the following features:

1. Ability to create a new Public/Private Key pairs (Wallet address)
2. A blockchain which maintains all the transactions and acts as a Ledger
3. Ability to update the blockchain and manage forks in the blockchain
4. Send and receive coins. Each coin can be split up to 6 decimal places.
5. Ability to check any wallet's balance, using its public key.
6. Ability to connect to other full nodes via the internet.
7. Miner Software, which will process all the unconfirmed transactions

_____, Committee Chair
Dr. Ted Krovetz

Date

ACKNOWLEDGEMENTS

I would like to give my recognition and appreciation to Dr. Ted Krovetz for taking up the role of my project advisor. He helped me understand the concepts in public key cryptography, cryptographic hashing and signature verification using elliptic curves; his timely feedback and guidance were immensely important for me, towards fulfilling the objectives set for the project.

In addition, I would also like to show my appreciation to Dr. Haiquan Chen for his willingness to serve on the committee. Lastly, I would like to thank Dr. Jinsong Ouyang and the entire faculty and staff of the Department of Computer Science Engineering at California State University, Sacramento, for their continuous administrative support.

TABLE OF CONTENTS	Page
Acknowledgements	vii
List of Tables	xii
List of Figures	xiii
Chapter	
1. INTRODUCTION	1
1.1 Overview	1
1.2 Features and Privileges	1
1.3 Prerequisites	4
1.4 Blockchain Technology	4
1.5 Entity Relationship Diagram.....	6
2. SOFTWARE DEVELOPMENT CYCLE	8
3. ANALYSIS - SYSTEM ARCHITECTURE AND DESIGN PATTERNS.....	11
3.1 Client-Server Architecture	11
3.1.1 Presentation Layer	13
3.1.1.1 Postman.....	13
3.1.2 Logic Layer.....	13
3.1.2.1 Node.js	13
3.1.2.2 RESTful API.....	13
3.1.2.3 Amazon EC2.....	14

3.1.3 Data Layer.....	15
3.1.3.1 MongoDB	15
3.1.3.2 Redis	15
3.2 Design Patterns	16
3.2.1 MVC Design Pattern.....	16
4. DESIGN.....	18
4.1 Transaction Structure	19
4.2 Block Structure	20
4.3 Proof-of-Work.....	22
4.3.1 Target Value for Proof-of-Work.....	24
4.3.2 Drawbacks of Proof-of-Work	26
4.3.2.1 51% Attacks	26
4.3.2.2 Power Consumption.....	28
4.4 Proof-of-Stake.....	28
4.5 The Ecash Consensus Algorithm.....	30
4.5.1 Coin-Age.....	30
5. IMPLEMENTATION.....	33
5.1 System Pre-requisites.....	33
5.1.1 Installing Node.js	33
5.1.2 Installing MongoDB	34
5.1.3 Installing Redis	36

5.1.4	Installing Postman.....	37
5.1.5	Running the Ecash Miner and Wallet Node	38
5.2	Ecash APIs	40
5.2.1	Wallet Key Generation API	41
5.2.2	Miner Login API	42
5.2.3	Get Miner's Current Coin-Age API	43
5.2.4	Get Latest Block API	44
5.2.5	Get User's Account Balance API	45
5.2.6	Create New Transaction API	46
5.2.7	Check Unconfirmed Transaction API	48
5.2.8	Non-API Background Activity	49
5.3	Code Implementation.....	49
5.3.1	Folder Structure	50
5.3.2	Package.json – Library usage	52
5.3.3	Configuration Files	54
5.3.4	Wallet Key Generation	61
5.3.5	Unconfirmed Transaction Generation.....	62
5.3.6	Verify and Store Unconfirmed Transaction Broadcasts	65
5.3.7	New Block Generation.....	67
5.3.8	Broadcasting Blocks and managing Forks in the blockchain	74
5.3.9	Coin-Age Calculations during Block Processing	76

5.3.10 Error Handling	78
5.3.11 Account Balance Calculations	79
5.3.12 Target Difficulty Calculations	81
5.3.13 Caching Account Balances for Performance	83
6. CONCLUSION AND FUTURE SCOPE	85
Bibliography	86

LIST OF TABLES

Tables	Page
1a. Features Provided by Miner and Wallet Nodes to its End-Users.....	3
1b. Features Provided by Miner and Wallet Nodes to its End-Users	4
2. Transaction Structure	19
3a. Block Structure.....	20
3b. Block Structure	21
4. Wallet Key Generation API.....	41
5. User Login API.....	42
6. Get Miner's Coin-Age API.....	43
7. Get Latest Block API.....	44
8. Get User's Account Balance API	45
9. Create New Transaction API	46
10. Check Unconfirmed Transaction API.....	48
11a. Modifiable configuration variables	56
11b. Modifiable configuration variables.....	57
11c. Modifiable configuration variables	58
11d. Modifiable configuration variables.....	59

LIST OF FIGURES

Figures	Page
1. ER diagram – eCash.js	7
2. Model of SDLC [2]	8
3. Input nonce's effect on block's hash value	23
4. Download Node.js with LTS	33
5. Node and NPM installation confirmation	34
6. Downloading latest version of MongoDB	35
7. Accessing MongoDB locally	35
8. Verify Redis installation	36
9. Downloading Postman	37
10. Postman application usage	38
11. Installing required libraries using NPM	39
12. Starting Ecash application	39
13. Wallet Key Generation API Sample Call and Response	41
14. Miner Login API Sample Call and Response	42
15. Get Miner's Coin-Age API Sample Call and Response	43
16. Get Latest Block API Sample Call and Response	44
17. Get User's Account Balance API Sample Call and Response	45
18. Create New Transaction API Sample Call and Response	46
19. Check Unconfirmed Transactions API Sample Call and Response	48

20. Folder Structure	50
21. package.json file	52
22. Constants defined within the application	55
23. Ports and Network Nodes Configuration File.....	60
24. Wallet key generation using the secp256k1 library	61
25. Supporting functions for Signature generation and verification	62
26. Unconfirmed Transaction Generation Function	63
27. Hashing Function for transaction generation	64
28. Adding and Broadcasting Unconfirmed Transactions	65
29. Validating Unconfirmed Transactions	66
30. Storing Unconfirmed Transactions into Redis.....	67
31. New Block Generation Function	69
32. Unconfirmed Transactions to Confirmed Transaction Array Function	70
33. Account Balance Validation for Transactions Function	71
34. Duplicate Transaction Removal Function	72
35. Block Hash Generation Function.....	73
36. Generated Block Broadcast Function	74
37. Receive Broadcasted Blocks from Network Function.....	75
38. Coin-Age Calculation Function	77
39. Error Codes for Invalid API Calls	78
40. Account Balance Calculation Function	80

41. Set Target Values for all Previous Blocks Function	82
42. Block Balance Caching Function for Coin-Age Calculation.....	84

1. INTRODUCTION

1.1 Overview

Ecash.js is a complete cryptocurrency developed using JavaScript allowing it to run on any device with minimal overhead. Ecash.js is capable of creating new user wallets to store coins, send and receive coins, check current balance for any wallet, a miner software which is responsible for validating, processing and confirming new transactions and the ability to bring it's distributed network towards a stable state using the blockchain technology and a specially developed Ecash consensus algorithm which is a combination of Proof-of-Work and Proof-of-Stake. All of Ecash's features will be provided as REST APIs which can be accessed using basic GET and POST requests to any that is any computer running the eCash.js Miner or Wallet software.

1.2 Features and Privileges

End-Users of eCash.js can access the features using two types of software: Wallet Nodes and Miner Nodes. The Wallet Node and the Miner Node have essentially the same software capabilities but are used for different purposes. The Miner Node is intended for automated use by generating new blocks and helping the network stay updated by broadcasting these new blocks and any forks in the blockchain. The Wallet Node on the other hand provides Application Programming Interface (API)

access to the End-Users, allowing users to send and receive coins and browser access to the blockchain eliminating the need for any special software.

Both the software users provide serve different roles and provide different privileges to its end-users, which are described in tables 1a and 1b below.

Table 1a. Features Provided by Miner and Wallet Nodes to its End-Users

Software Features		Miner Node (No End-User Access)	Wallet Node (End-User Accessible)
User	Login	✓	X
	Get Users Coin Age	X	✓
	Get Users Balance	X	✓
Wallet	Generate New Public/Private Key Pair	X	✓
Transaction	Create New Transaction	X	✓
	Get Unconfirmed Transaction	✓	X
	Broadcast Unconfirmed Transaction	✓	✓
	Accept Broadcasted Unconfirmed Transaction	✓	X

Table 1b. Features Provided by Miner and Wallet Nodes to its End-Users

<div> <div>Software</div> <div>Features</div> </div>		Miner Node (No End-User Access)	Wallet Node (End-User Accessible)
Block	Get Block by Id	X	✓
	Get Latest Block	X	✓
	Create New Block	✓	X
	Broadcast Block	✓	✓
	Accept Broadcasted Blocks	✓	✓
	Update Blockchain on Node Restart	✓	✓

✓ - The feature is available to the End-Users.

X - The feature is available, but not recommended for End-User access.

1.3 Prerequisites

This application uses Node.js which is a server-side version of JavaScript. End-Users can access the application with RESTful Web APIs. To begin development, the developer must have prior technical knowledge of Node.js (JavaScript) and a

non-relational database like MongoDB. In this document, I am providing enough information about the standard development resources which are available freely online, to help the readers better understand the development tools used while developing this application. As this project mainly focuses on the backend Node.js development, it does not cover technical details about the frontend. However, having some knowledge of the frontend technologies like jQuery will help the reader with integrating Ecash.js as a mode of payment for any website or any application with internet access.

The development tools used are stated below:

1. Node.js (for backend end development)
2. Postman (for frontend request simulation)
3. RESTful API development with Express framework
4. MongoDB Database
5. Redis as Memory Cache
6. Sublime Text or any other Text Editor

1.4 Blockchain Technology

The blockchain technology is generally used as a permanent data-store over a decentralized distributed network. As the name suggests blockchain is a basically a chain of blocks similar to a linked list with one simple addition. The key difference between a linked list and a blockchain is that the blocks in the blockchain contain the hash value of its preceding block. This simple addition causes malicious

alterations to any single block affect all the subsequent blocks due to the cascading change in the “previous block hash” values in every block following the modified block. Any machine containing a live up-to-date copy of the blockchain is called a Full-Node. Sometimes the blockchain can have 2 versions generated parallelly due to the network delays between the geologically separated Full Nodes around the world, this situation is called a fork. The blockchain will use the consensus algorithm to determine the valid one of these branches and all other nodes will discard the invalid version. Forks in Ecash generally differ by 2-3 blocks.

1.5 Entity Relationship Diagram

An entity relationship diagram is a graphical representation of entities and their relationships with each other. A relationship is how the data is shared between entities and it is an organization of data in databases [1]. Figure 1 shows the entity relationship diagram for the Ecash. It’s important to note that this project uses a blockchain, hence its data storage structure is straight-forward, especially with the use of a non-relational database like MongoDB.

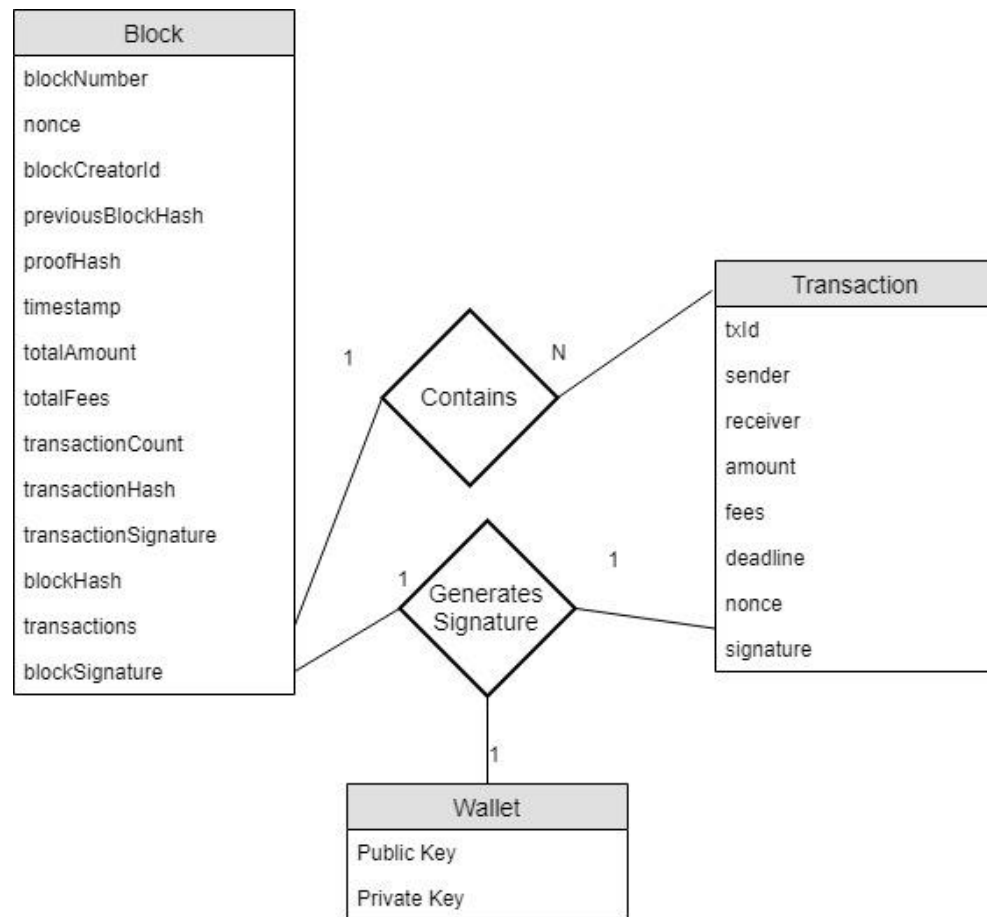


Figure 1. ER diagram – eCash.js

2. SOFTWARE DEVELOPMENT CYCLE

In this chapter, there is a brief description about the Software Development Life Cycle (SDLC), and it shows a path from planning to deployment. There are different types of SDLC models, and each SDLC model is divided into phases. There is a task associated with each phase, as shown below in Figure 2, which are Planning, Analysis, Design, Implementation, and Maintenance [2].

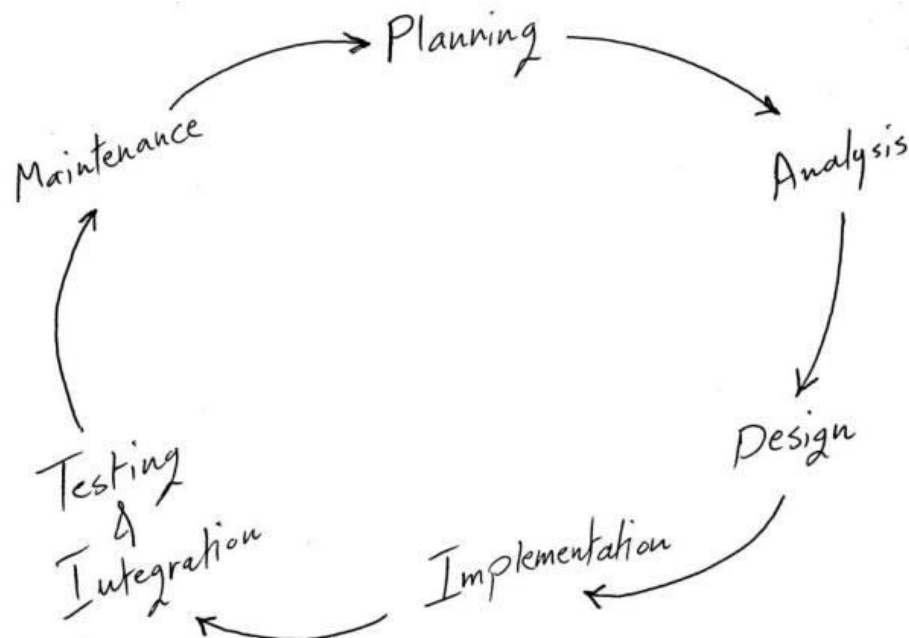


Figure 2. Model of SDLC [2]

1. Planning – The Planning phase is the most crucial step in creating a successful system, during this phase we decide exactly what we want to do and the problems we’re trying to solve. In Ecash, the key requirement is ease of use with minimal overhead. This has

been achieved using Node.js and providing REST API end-points, so users can access the Ecash network with any web browser.

2. Analysis – In this phase, we must analyze the parameters from the planning stage. We must decide how efficiently the product could be developed using different architecture and design patterns. In Ecash, the client-server architecture is used for development of the application.
3. Design – In this phase, developers need to decide how the end-users access the application. Due to the RESTful implementation of Ecash the client only needs a web browser and can access the currency with simple webpages or applications with internet access.
4. Implementation – This phase comes after obtaining a complete understanding of system requirements and specifications, we begin with the actual code implementation for the required features. The development of this phase is dependent on the 3 phases discussed as above.
5. Maintenance – This is the last phase of our SDLC. In this phase, periodic maintenance for the system needs to be carried out to prevent the system from becoming obsolete. This will include software changes to better utilize newer hardware and continuously evaluating the existing system's performance. It also includes providing latest updates for certain components to make sure the application uses latest technologies security patches. It is important to note that changing the Block or Transaction structure at a later point is not a feasible approach, thus we must design them thoroughly.

The previous chapters provide a brief description about the Ecash application. While this chapter describes the planning and requirement gathering phase. The following chapters will provide a detailed discussion about the Analysis, Design, and Implementation phases for Ecash.js. During the Maintenance phase, several testing cycles were conducted for the Ecash network. Any bugs found were analyzed and resolved immediately.

3. ANALYSIS - SYSTEM ARCHITECTURE AND DESIGN PATTERNS

In this chapter, we will discuss the Analysis phase of Ecash.js. Let's start with a brief introduction to the client-server architecture.

3.1 Client-Server Architecture

“Client-server architecture is a computer network in which many clients (End-Users) request and receive service from a centralized server (Wallet/Miner Node). Client devices provide an interface to allow a user to request services of the server and to display the results the server returns. Servers wait for requests to arrive from clients and then respond to them. Ideally, a server provides a standardized transparent interface to clients so that clients need not be aware of the specifics of the system (i.e., the hardware and software) that is providing the service. This computing model is especially effective when clients and the server each have distinct tasks that they routinely perform” [3].

In Ecash, an example of the client device is a webpage capable of creating new transactions and checking user balances while the server is a computer running the Miner Node or Wallet Node which manages the database where the live up-to-date blockchain is permanently stored. Many clients can access the server's information simultaneously, each Full Node can receive requests independently and is capable of serving those requests independently.

It is important to note that even though we are using a client-server architecture for the End-Users, this application is still a distributed application at its core due to the consensus algorithm implemented in each server. The End-User can use an existing Full Node to send and receive coins and check account balances. An End-User who does not trust other Full Nodes can host his own Miner Node or Wallet Node and access it from any light-weight device like a smartphone.

The 3-tier architecture is a client-server architecture in which the functional process logic, data access, computer data storage, and user interface is developed and maintained as independent modules on separate platforms [4].

The three tiers/layers in a three-tier architecture are:

1. Presentation Tier: It occupies the top level and displays information related to services available on a website. This tier communicates with other tiers by sending results to the browser and other tiers in the network [4].
2. Application Tier: It is also called the middle tier, logic tier, or business logic. This tier is pulled from the presentation tie, and it controls application functionality by performing detailed processing [4].
3. Data Tier: It houses database servers where information is stored and retrieved. Data in this tier is kept independent of application servers or business logic [4].

3.1.1 Presentation Layer

The Wallet Node responds with JSON objects to all API requests.

3.1.1.1 Postman

The presentation layer for my application was developed with Postman. Postman is a tool capable of sending basic GET and POST requests to our Wallet Node. The Ecash payment system can be integrated into any existing website or application by simply accessing the APIs as demonstrated with postman in the following chapters.

3.1.2 Logic Layer

3.1.2.1 Node.js

Node.js applications are written using JavaScript that runs on any computer using the Chrome V8 engine. It is platform independent and can run on Windows, Mac and Linux. Using Node Package Manager (npm) we can use multiple readily available libraries to accomplish the objectives of this project.

3.1.2.2 RESTful API

Node Package Manager can be used to install Express. Express is used to open API endpoints to our application. A RESTful API is an application program interface (API) that uses HTTP requests to

GET, PUT, POST and DELETE data. A RESTful API breaks down a transaction into a series of small modular steps. Each module addresses a particular underlying role in completing the transaction.

3.1.2.3 Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides secure, resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers [7].

Amazon EC2's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay only for capacity that you actually use. Amazon EC2 provides developers the tools to build failure resilient applications and isolate them from common failure scenarios [7].

3.1.3 Data Layer

3.1.3.1 MongoDB

We will be using MongoDB as it is a non-relational database, where we can store the blocks for the blockchain as-is. Given that each block can have a varying number of transactions, a non-relational approach is well suited for this application.

3.1.3.2 Redis

Redis is a key-value pair database which stores data in the computers main memory. We have used it to cache the Wallet key for the Node owner and to store all unconfirmed transactions. Hence if any node shuts down abruptly, it will lose all of its unconfirmed transactions and will need to fetch them from other Full Nodes.

3.2 Design Patterns

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system [5].

3.2.1 MVC Design Pattern

Since blockchain is a relatively new technology, there are no well-established design patterns for it, thus I will be using a modification of Model-View-Controller (MVC) design pattern. This pattern is used to separate application's concerns.

Model - Model represents an object carrying data. It can also have logic to update controller if its data changes. Since we are using a blockchain, there will be no changes to a data block, once it has been forged, unless there is a fork in the blockchain.

View - View represents the visualization of the data that model contains. All objects are accessible as JSON objects.

Controller - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate [6].

We will use the unconfirmed transaction objects to create a new block in the Miner Node.

However, since we are using Node.js which is not an object-oriented language, all objects specified above will be assessed through functions and be represented as JSON objects. The MVC design is only responsible for the internal working, as the external APIs are near real-time, they do not affect the internal working of an Ecash Full Node in anyway.

4. DESIGN

We will need to design the Transaction structure and the Block structure before we begin with development. These object structures are important, as we cannot modify these at a later stage – any modification after initial deployment, say when the network Nodes are at block number 10,000 will require us to keep the existing Block Structure code intact for new Full Nodes to verify blocks older than block number 10,000 while adding new functions to handle blocks generated after block number 10,000. Thus, any modification to the block and transaction structure will lead to redundancy in the application code and will make the application increasingly difficult to maintain.

After some analysis, I decided having the following fields in the Transaction and Block objects, as described in tables 2, 3a and 3b below.

4.1 Transaction Structure

Table 2. Transaction Structure

Transactions	
Field	Represents
txId	SHA-256 hash of the entire transaction object, used to uniquely identify any transaction
sender	Coin senders Public key
receiver	Coin Receivers Public key
amount	Amount of coins being transferred in this transaction. Each coin can be divided upto 8 decimal places
fees	Fee the sender is willing to pay to the Miner for validating and processing this transaction
deadline	The block number, upto which this unconfirmed transaction is valid. If this transaction is not confirmed by the specified deadline block number, the transaction will be discarded from the Miner's list of unconfirmed transactions.
nonce	A random integer added as salt to the transaction. This number must be greater than 1 million.
signature	Every transaction needs to be signed by the sender, to authenticate and prove the genuineness of this transaction

4.2 Block Structure

Table 3a. Block Structure

Blocks	
Field	Represents
blockNumber	Unique serially incremented number which represents the blockNumber th block in the blockchain starting from 0
nonce	A random integer added as salt to the block. This number must be greater than 1 million. Nonce can be changed to attempt obtaining a valid proof hash value while Mining for a new block
blockCreatorId	Block Miner's Public key
proofHash	Proof Hash proves a certain amount of computational effort was put into mining this block. It's the SHA-256 Hash of the current block with a random nonce. If this hash is valid, the block mining process is assumed to be complete and the new block is broadcasted over the network
timestamp	Current timestamp in Unix format
totalAmount	The combined total amount of coins being transferred in all the transactions present in this block

Table 3b. Block Structure

Blocks	
Field	Represents
totalFees	The combined total amount of coins being payed to the block creator as transaction processing fees by all the transactions present in this block
transactionCount	Total number of transactions confirmed in this block
transactionHash	A cumulative hash of all the transaction's hash values
transactionSignature	transactionHash signed using block creator's private key
transactions	An array of all transactions confirmed in this block
blockHash	SHA-256 hash of the complete block object
blockSignature	blockHash signed using block creator's private key
previousBlockHash	Hash value of previous block in the blockchain. This provides security to the blockchain, as any modification to one of the previous blocks, will cascadingly affect the previousBlockHash values for every existing block after the modified block

Note: SHA-256 is the only hash function used throughout the project. Assume all hashes are SHA-256 if not specified.

Having finalized the block structure and the transaction structure, we need to design an algorithm to bring the distributed system into a state of consensus without

depending upon a centralized authority. This can be done in 2 ways – Proof-of-Work and Proof-of-Stake as discussed below. I have combined both of these approaches towards developing a special consensus algorithm for Ecash.

4.3 Proof-of-Work

Proof-of-Work was first used in Bitcoin for consensus. The basic idea for Proof-of-Work – as the name suggests – is to prove the miner (block creator) put in a certain amount of computational effort into mining the block. This computational effort also needs to be verifiable over a distributed system without a centralized authority. The way I have implemented this is using a targeted hash value. SHA-256 creates a statistically unique hash value for any input sequence. A good approach to distributed consensus in the network is to find a block hash value that starts with a pre-determined number of zero bits. I have defined it as `proofHash` in my block object structure. Since a single bit change in the block object changes its output hash value drastically, we can use a random integer – in this case called `nonce` (number used once). We can keep trying different `nonce` values in the block until we find a hash value that starts with the target number of zero bits.

```

Nonce : 165621 Hash : 50dbf6510b9faa66e6495faf3941b402ee64bf96bf619d9b2d9e97612ebf8be4
Nonce : 165622 Hash : a50ebe2d7b02256f6f03634ac7c3ba8943a1f348fcef0365715290761b039b1f
Nonce : 165623 Hash : 4ee28907ac3704f85322fd807293f04fa1fb9f0f7e78f5937743e17965e29e71
Nonce : 165624 Hash : 9097ca7bc1b21aea0067e17aa523b7e6e6bfc9219437e5d9b35a88673d8faa5f
Nonce : 165625 Hash : 390fcf38d79644e1f516143fdacc627972de0611e272f6cfb6cef39a242207c0
Nonce : 165626 Hash : 55fdf517b3082d196211e41fc22f7b528bf8de06b3d04dbe0542581dc1392466
Nonce : 165627 Hash : 03ea3d897ddce389d37e920b75e111abf1692b12340abb859a5226cecbc5cd0e
Nonce : 165628 Hash : 5cbbf4e865c2b8c0437f9f7d5fd6c06fe08d3c16a76a7c55c0be1ba7a414016e
Nonce : 165629 Hash : 952670ba90f783c1d0a3d83b317216459138a91f04660c2f671d15c41d12d394
Nonce : 165630 Hash : e6300a4906ef717d4f2772a781e5837e209120582b9bad78b9df24773caac760
Nonce : 165631 Hash : 3059597794f500f050e2b85cd0c9bd24ba1e17a40e92fc78ea7f0604602751f4
Nonce : 165632 Hash : f742b649fa6d1d178b6f9f4e266ecf84fd888dc6958d5a31d215867341d48c5b
Nonce : 165633 Hash : 618e42b22d61f65adca71ea313b96ae465433ca9740a5b1a6b1f98b8d043f167
Nonce : 165634 Hash : ed2ffea707558c2e90d2b84e1ea73817eaf23db86de9b48cb62d9305d0c941cc
Nonce : 165635 Hash : 8885277632d719eeb24ca662965960dee3c472af7529debe17c2e031ee5aaf94
Nonce : 165636 Hash : d7301c7cc833229ce2fd17738dbe5862d6098380fd892a6f9779e7f4af076648
Nonce : 165637 Hash : 19c1fb31e26712b7bc40113d4176e322f5420e5213356b01d5ffe40fcb73b35e
Nonce : 165638 Hash : f49683b05a622d0a4d3915bd80ecf1846299d95f7c3820bb0ad4711ced51b9a2
Nonce : 165639 Hash : abc096926b9291cd5ec71ef9fd4ef0ce15dd5099d8cb9b21b00c71851357fba5
Nonce : 165640 Hash : d27d25e18791ce0981de51095f3d7ecbe218a2693aae933b1c60456a239dbccce
Nonce : 165641 Hash : c97daea80a43849947c3b7835994e44da159b46bfa04dde2f1188e732c0bf5b9
Nonce : 165642 Hash : 2d2c4e0e5dbc86f8892abb70b136d5b2409af9dc135e1e72ff3c15c7cdd78fd6
Nonce : 165643 Hash : 07f4563b58bd6e476069e3465da03ec87ea512ffcadbbcb4e8c0cf248e2324b87
Nonce : 165644 Hash : 2cc05277a33a936ab60819aaaf62ca8ff7816e0174bf3c417f7c5ef99c9b5208
Nonce : 165645 Hash : e45b863ae0c624866cb0fa68b5174d5f2983bb16a6b980ffc5977741c7e9bf92
Nonce : 165646 Hash : ae9e16080f7cf9cea016732cf0a19b3c9312b30fe799e0433dd9647a7f715f60
Nonce : 165647 Hash : 1187536c22d685e7e5a41656b6bde99d89f24aa0fa4279882caf28c198aa6e44
Nonce : 165648 Hash : 26cb3b6b25bbf8b1e6c93e504bf1a68e0d1907b8e3d4217c83f1d4c45ed99499
Nonce : 165649 Hash : 6162f800308c49a57c22e3e0942c7f0237b4db3953a241cb95e92491f5d5b50f
Nonce : 165650 Hash : 00000dd94704ca22e869a240fba90fb9ab7ef079c7eb1ec8c3a30a209320acb4
BlockGenerationTime: 29190.295ms
Incoming Broadcast Block : {
  "blockNumber": 560,
  "nonce": 165650,
  "blockCreatorId": "035a30203ad6e727794f2e2748668831359e0957fd343940ddf0d1c44057821171",
  "previousBlockHash": "e30ebcb5dfddbfa59a10eb3b91ad5a84db243b70de4b84412f614f2da09abefe",
  "proofHash": "00000dd94704ca22e869a240fba90fb9ab7ef079c7eb1ec8c3a30a209320acb4",
  "timestamp": 1512578600838,
  "totalAmount": 0,
  "totalFees": 0,
  "transactionCount": 0,
  "transactionHash": "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
  "transactionSignature": "e00926a7258034e0911fbd89051137cbab16c99dea8f924cd710806180e9f78",
  "blockHash": "e4cd5db99d4babe18a4c1ec19092b48d1312ba65cda375a1c3c9e9092dc8e38f",
  "blockSignature": "317e6caff039351b83e4bee5e84787f585f19e238348bae46c462d2a6d5caf2d6f152",
  "transactions": []
}
Block Done !
^C

```

Figure 3. Input nonce's effect on block's hash value

As we can clearly see in Figure 3, a single increment to the nonce value completely changes the hash value for the block. We keep trying new numbers until we get the desired zero bits in the proofHash. In Figure 3 we were trying to find a nonce value which generates a proofHash beginning with 20 zero bits (5 hexadecimal zeros) and we broadcast the block as soon as we find a valid proofHash.

Proof-of-Work can be thought of as a computational race between all the Miner Nodes in the world currently active on the network. The first miner to successfully find a target satisfying proofHash wins the block and receives the block mining reward and all the fees associated with each transaction in the block.

Having discussed the basic structure of the Proof-of-Work algorithm implemented in Ecash, we need to determine a suitable target value for mining blocks, I will explain this in the next section.

4.3.1 Target Value for Proof-of-Work

Before we design the target value calculation logic, we need to understand why it's important in the Proof-of-Work process. The main objective of the target value is to converge the block mining time for the next blocks towards a pre-decided time value. For Ecash, I have decided to keep the average time for block generation as 1-minute. If we reduce the block generation time any further, the network delays between geologically separated Miner Nodes will cause the blockchain to have multiple parallelly growing branches in different parts of the world, until they converge. This means the

branch that won will replace all the other blocks in the losing blockchain making the blockchain unreliable for a large number of blocks. Increasing the time too much implies that each block will take more time to Mine, meaning the transaction processing speed will reduce significantly since each block can process a configured maximum of 512 transactions.

When we deploy the Miner application for the first time there will only be a few Miner Nodes running over the network, hence the computational power of this network will be limited. But as more and more Miner Nodes register onto the network the computational power will increase exponentially. Hence the target difficulty needs to increase to bring the block mining time back to the pre-decided 1-minute average.

When we increase the target value by a single extra zero bit, we essentially double the difficulty for the network. For example, SHA-256 generates a 256-bit hash output and we need the first 4 bits to be zero bits – we will need to try approximately $O(2^4)$ random numbers until we find a solution.

When change the requirement to a hash value starting with 5 zero bits, the difficulty increases exponentially requiring approximately $O(2^5)$ attempts.

It is also important to note the dynamic nature of the Miner Nodes. They can connect and disconnect from the network freely without any warnings, hence the target value will decrease to reduce the Mining difficulty of the network when too many Miner Nodes disconnect from the network.

Using the logic discussed above, the target value maintains the network's Mining time at 1-minute on average. A new target is calculated after every 1000 blocks by finding the average time required to generate the previous 1000 blocks and increasing or decreasing the network difficulty accordingly to bring the average time closer to 1-minute.

Both of these values – 1-minute and 1000 blocks – can be changed in the configuration file but should not be changed once the initial node has been deployed. After a lot of testing, I have decided to use 1-minute averages every 1000 blocks since these values balance the network approximately every 1000 minutes (between 16-17 hours), that is at least once every day.

Having discussed the internal logic behind the implemented Proof-of-Work algorithm implementation, I will now explain some of drawbacks of this approach.

4.3.2 Drawbacks of Proof-of-Work

4.3.2.1 51% Attacks

The first drawback of the Proof-of-Work algorithm is called the 51% attack. Here the attacker gains control of 51% or more of the Miner Nodes either by malicious means or by suddenly introducing new Miner Nodes to the network (greater than or equal to the number of existing Miner Nodes) to finally obtain 51% of the network. Once at this stage, the attacker has a 51% chance of mining the next block

and 100% chance of parallelly mining the existing blocks. He basically keeps mining the exact same blocks as the main network but does not broadcast them. He can then for example pay a merchant a certain number of coins on the main network but will exclude this transaction from his parallelly running network. Once the attacker receives the products or services from the merchant, he will broadcast his version of the blockchain to the network by mining the next block before the network. Now all the nodes on the network will update their blockchain to the attacker's version since it's the largest blockchain available. The End-Users will not realize this since their transactions are preserved and only the attacker's transactions have been removed. This is called a double spend attack where the attacker is spending the same coin multiple times. Double spending will completely break the payment system.

It is also important to note that the attacker does not broadcast any of the parallelly Mined blocks onto the network hence he does not compete with the network nodes until he wants to attack the network. Thus, the attacker can attack the network any number of times without increasing the Mining difficulty of the network. The Mining difficulty will only increase if the attacker tries to continuously publish new blocks onto the main network.

4.3.2.2 Power Consumption

The only way to stop the 51% attack as mentioned above is to keep increasing the number of genuine Miner Nodes on the network. This is not a sustainable permanent solution since each Miner Node – being at 100% load for the most part – will consume a large amount of electricity. Given 80% of the current electricity generation is done via burning coal, increasing the number of nodes will indirectly harm the environment, while increasing the transaction fees expected by the Miners due to increased competition and the increasing electric bills.

Another approach for bringing the network to consensus is Proof-of-Stake as discussed below.

4.4 Proof-of-Stake

Proof-of-Stake gives every coin holder a fair chance to mine the next block based on the number of coins they hold. Hence, users of a Proof-of-Stake based cryptocurrency will be incentivized for holding their coins rather than holding the physical machines Mining the next blocks.

This gives better protection against 51% attacks since the attacker would need to obtain 51% of the cryptocurrency to carry out a 51% attack. Proof-of-Stake avoids this ‘tragedy’ by making it disadvantageous for a miner with a 51% stake in a

cryptocurrency to attack the network. Although it would be difficult and expensive to accumulate 51% of a reputable digital coin, a miner with 51% stake in the coin would not have it in his best interest to attack a network in which he holds a majority share. If the value of the cryptocurrency falls, the value of his holdings would also fall, and so the majority stake owner would be more incentivized to maintain a secure network [8].

But this approach has a huge disadvantage. We may argue that Proof-of-Stake is not an ideal option for a distributed consensus protocol. One issue that can arise is the "nothing-at-stake" problem, wherein block generators have nothing to lose by voting for multiple blockchain histories, thereby preventing consensus from being achieved. Because unlike in Proof-of-Work systems, there is little cost to working on several chains [9].

Proof-of-Stake has also never been tested over a long time like Proof-of-Work and Proof-of-Stake has many variations to the Proof-of-Stake algorithms.

Hence, I have used a combination of Proof-of-Work and Proof-of-Stake to reduce the chances of 51% attacks while still being mined primarily with Proof-of-Work.

4.5 The Ecash Consensus Algorithm

For Ecash I have combined these 2 consensus algorithms into a single consensus algorithm. This combination mainly aims to reduce the chances of a 51% attack without losing the time-tested security of Proof-of-Work.

As discussed in the Proof-of-Stake section, Ecash Miners need to hold coins for a certain number of blocks, before they become “stake-able” coins. A stake-able coin can be thought of as a voting ticket. The larger number of stake-able coins a Miner holds, the higher his chances of winning the next generated block. We calculate “Coin-Age” using these stake-able coins. Coin-Age gives us a single number for each competing Miner, which can be compared in a straight-forward manner to determine the winner.

4.5.1 Coin-Age

Coin-Age can simply be defined as the currency amount multiplied by the holding period. In a simple to understand example, if Bob received 10 coins from Alice and held it for 90 blocks, we can say that Bob has accumulated 900 coin-blocks of Coin-Age.

Additionally, when Bob spent the 10 coins he received from Alice, we can say the Coin-Age Bob accumulated with these 10 coins had been consumed (or destroyed).

I have used the following formula to calculate Coin-Age:

$$\text{Coin Age} = \sum_{k=\text{currentBlock}-\text{maxStakeBlocks}}^{\text{currentBlock}} \text{Number of Stakeable Coins in Block } k^i$$

Number of Stake-Able coins in the equation above represents the number of coins in the k^{th} block that have been held (unspent) for more than the defined minimum stake-able hold blocks count. This has been set to 20 blocks and is configurable in the configuration file.

To prevent small number of individual Miners from gaining massive Coin-Age, I have set a hard limit of 100 blocks to be considered for the stake calculation. This limit is configurable in the configuration file but should not be changed once the blockchain has been initialized. I have set them to small numbers to perform faster testing.

The Miner with higher Coin-Age will be given higher preference, and one coin's stake is consumed per block generated. As discussed in the disadvantages of Proof-of-Work an attacker would need to generate a large chunk of the blockchain parallelly to reverse a transaction, which would mean an attacker would need to Mine many blocks before reaching the current block hence their stake will keep on reducing with each block they Mine making it harder and harder to perform the 51% attack as time goes by. This would lead the attackers to only be able to attack the last few blocks depending on the amount of coins they hold. Any subsequent attack attempts will not be feasible since they will consume a good amount of their stake (Coin-Age) for each attack and need to wait for the defined minimum stake-able hold blocks count before their new coins start accumulating stake. Hence a higher number of hold blocks will lead to higher security.

Having discussed the design of the Transaction and Block objects in the blockchain and analyzed the consensus algorithm responsible for bringing the Ecash distributed network in a single stable state without requiring a centralized authority, we can now proceed to the implementation details in the next chapter.

5. IMPLEMENTATION

Implementation has been done using JavaScript to promote usage on any device with minimal overhead. Here I'll explain how to get started with Node.js and setting up the environment for running a Miner or Wallet Node.

5.1 System Pre-requisites

Here we will cover installing Node.js, MongoDB, Redis, Postman on your system and running the Ecash Miner and Wallet applications.

5.1.1 Installing Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. As an asynchronous event driven JavaScript runtime, Node is designed to build scalable network applications [10].

The setup is quick and easy. Just choose the correct operating system in the LTS section and download the official setup files from <https://nodejs.org/en/download> and follow instructions in the setup menu.

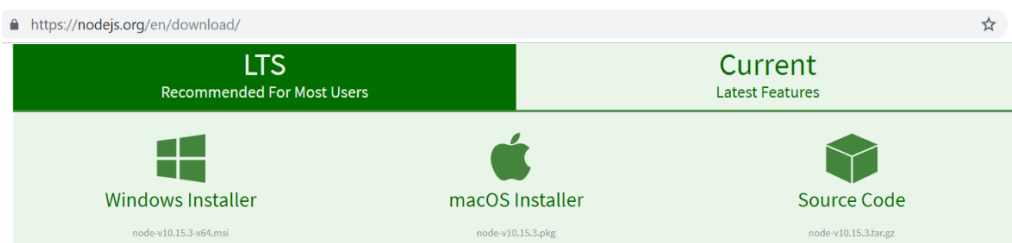
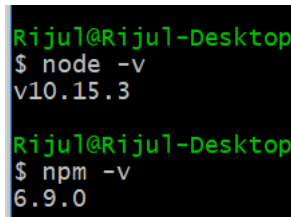


Figure 4. Download Node.js with LTS

The setup also contains the Node Package Manager (npm) which can easily install all required libraries for any Node.js project. All required libraries should be added to the package.json file in the project's root directory. Verify installation using “node -v” and “npm -v” commands in your preferred command line interface.

A terminal window with a black background and green text. The prompt is 'Rijul@Rijul-Desktop'. The first command is '\$ node -v' which returns 'v10.15.3'. The second command is '\$ npm -v' which returns '6.9.0'.

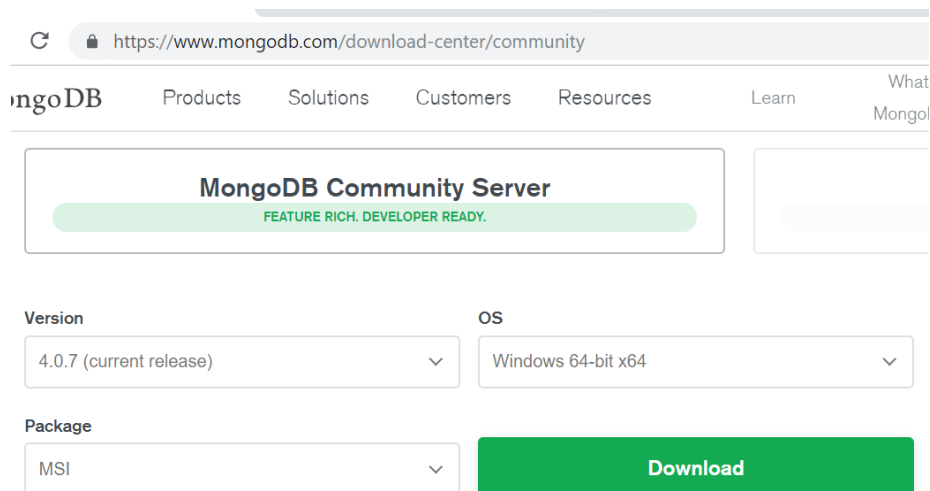
```
Rijul@Rijul-Desktop
$ node -v
v10.15.3
Rijul@Rijul-Desktop
$ npm -v
6.9.0
```

Figure 5. Node and NPM installation confirmation

5.1.2 Installing MongoDB

MongoDB is a document database with the scalability and flexibility that you want with the querying and indexing that you need. MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time [11].

Choose the correct operating system on MongoDB's official website <https://www.mongodb.com/download-center/community> and download the current release for MongoDB and follow instructions in the setup menu.



The screenshot shows the MongoDB download center page. The URL in the browser is <https://www.mongodb.com/download-center/community>. The page features a navigation bar with links to Products, Solutions, Customers, Resources, Learn, and What's New. Below the navigation bar, there is a section for the MongoDB Community Server, highlighting it as 'FEATURE RICH. DEVELOPER READY.' Below this, there are three dropdown menus for selecting the version, OS, and package type. The version is set to 4.0.7 (current release), the OS is Windows 64-bit x64, and the package type is MSI. A green 'Download' button is visible next to the package type dropdown.

Figure 6. Downloading latest version of MongoDB

Start MongoDB by running “mongod.exe” for Windows or the command “mongod” for Linux and MacOS. To check if MongoDB is running, type “mongo” in your preferred terminal or open “mongo.exe”. An output similar to Figure 7 should appear.

```
MongoDB shell version v3.4.10
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.10
Server has startup warnings:
2019-03-25T20:17:34.592-0700 I CONTROL [initandlisten]
2019-03-25T20:17:34.592-0700 I CONTROL [initandlisten]
2019-03-25T20:17:34.592-0700 I CONTROL [initandlisten]
nrestricted.
2019-03-25T20:17:34.592-0700 I CONTROL [initandlisten]
>
```

Figure 7. Accessing MongoDB locally

Please note the default port number used by MongoDB is 27017 and can be changed in the “mongod.conf” file. If this port is changed, we will also need

to make appropriate changes in the Ecash project configuration files. I'll discuss the project configuration files in the code implementation section.

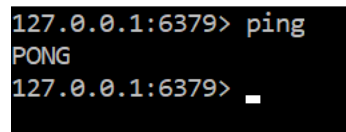
5.1.3 Installing Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, geospatial indexes with radius queries and streams [12].

Download the latest Redis version installation for Windows from <https://github.com/dmajkic/redis/downloads> or download the official files and follow instructions at <https://redis.io/download> for Linux and MacOS.

Being an in-memory key-value pair database, Redis is extremely lightweight and should be installed with ease.

Check if Redis is running in the background by running “redis-cli.exe” on windows or “redis-cli” command on Linux and MacOS. It should display a terminal similar to Figure 8. You can type “ping” and press return, the Redis server should respond with “pong” if it has been installed properly.

A terminal window with a black background and white text. The prompt is '127.0.0.1:6379>'. The user has entered 'ping' and the server has responded with 'PONG'. The prompt is now '127.0.0.1:6379> _' where the underscore indicates the cursor position.

```
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> _
```

Figure 8. Verify Redis installation

Note: the default port for Redis is 6379 and can be changed in the “redis.conf” file in the Redis installations root folder for Windows or “/etc/redis/6379.conf” for Linux users. If this port is changed, we will also need to make appropriate changes in the Ecash project configuration files.

5.1.4 Installing Postman

Postman is a tool that can send RESTful HTTP GET, POST, PUT, DELETE requests. We will use Postman as a frontend to access the APIs provided by Ecash in this project. Any website or application can easily implement the Ecash payment system into their product by referring to the Postman examples given in the section 5.2.

Similar to the previous installations, we will download the setup from postman’s official website <https://www.getpostman.com/downloads/> and select the appropriate operating system.

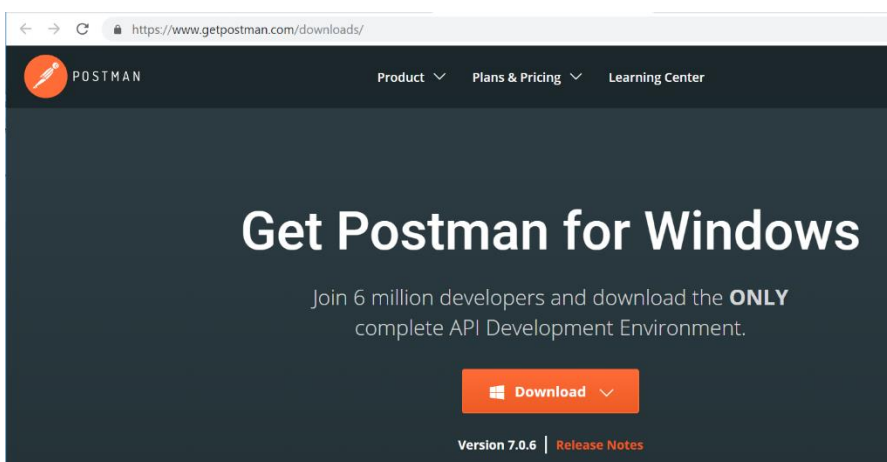


Figure 9. Downloading Postman

After installation completes, you can directly access postman through the desktop shortcut. You should see a screen similar to Figure 10 below.

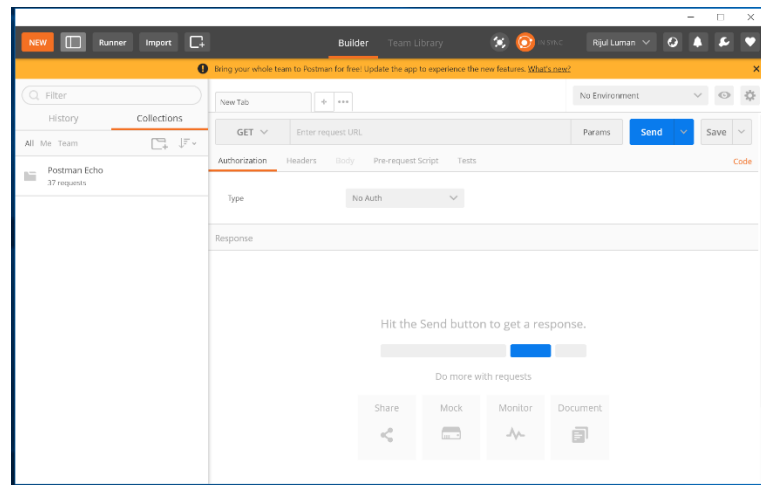


Figure 10. Postman application usage

Having the basic environment ready, we can proceed with running the Ecash application.

5.1.5 Running the Ecash Miner and Wallet Node

After setting up the environment as discussed in the previous steps, we will need to install all the libraries required by the project. To do so, we simply run “npm install” in the root directory of the eCash project folder. It will download and install all the required libraries from the internet as shown in Figure 11 below.

```
Rijul@Rijul-Desktop MINGW64 /d/My Folder/Co
ter)
$ npm install
npm WARN eCash@1.0.0 No repository field.

audited 413 packages in 0.936s
found 0 vulnerabilities
```

Figure 11. Installing required libraries using NPM

We can now run the Ecash Miner and Wallet Nodes by simply running the following commands in the project directory on your preferred terminal: “node walletServer.js” to run the Wallet Node and “node miner.js” to run the Miner. It may take several minutes to update the local blockchain to the current blockchain available on the network nodes specified in the config file.

```
Rijul@Rijul-Desktop MINGW64 /d/My Folder/College/CSUS/Project/Masters/eCash (n
ter)
$ ls
AddMongoIndex.js  config/  miner.js  package.json  README.md
app/              extra/  node_modules/  package-lock.json  walletServer.js

Rijul@Rijul-Desktop MINGW64 /d/My Folder/College/CSUS/Project/Masters/eCash (n
ter)
$ node walletServer.js
Mongo DB Started

Rijul@Rijul-Desktop MINGW64 /d/My Folder/College/CSUS/Project/Masters/eCash (n
ter)
$ node miner.js
Mongo DB Started
```

Figure 12. Starting Ecash application

Note: both the Miner and the Wallet Nodes cannot run parallely on the same machine since they are set to use the same port for the API endpoints

by default. To run them parallelly we simply need to change the ports for the Wallet Node.

If the environment was setup correctly using the default ports for MongoDB and Redis, you should not see any errors when you run the application.

5.2 Ecash APIs

Both the Miner and Wallet Server will provide access to all the APIs. But it is recommended to use the Wallet Nodes for End-User requests like creating new transactions or finding the latest or specific blocks and getting user balances' in the current blockchain. In this section, I'll provide the API access URL, a sample of the expected input (if any) and sample outputs. All the inputs and outputs to these HTTP APIs will be JSON objects.

5.2.1 Wallet Key Generation API

Table 4. Wallet Key Generation API

Request Type	GET
API End-Point	/api/key/generate
Sample URL	http://localhost:3000/api/key/generate
Input	<none>

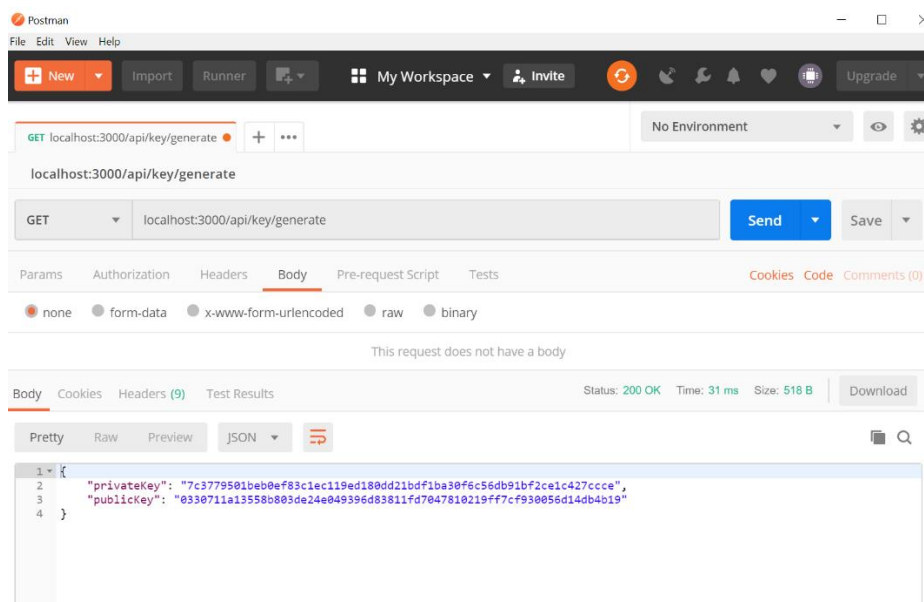


Figure 13. Wallet Key Generation API Sample Call and Response

This API can be used by the End-User to generate a new public-private key pair. This key pair can be used to send and receive coins over the Ecash network. API access information given in Table 4. The user is supposed to store the private key in a safe place and use it for future transactions. If the user loses his private key, there is no way to recover the coins in his wallet.

5.2.2 Miner Login API

Table 5. User Login API

Request Type	POST
API End-Point	/api/user/login
Sample URL	http://localhost:3000/api/user/login
Input	JSON object containing a public key and its private key

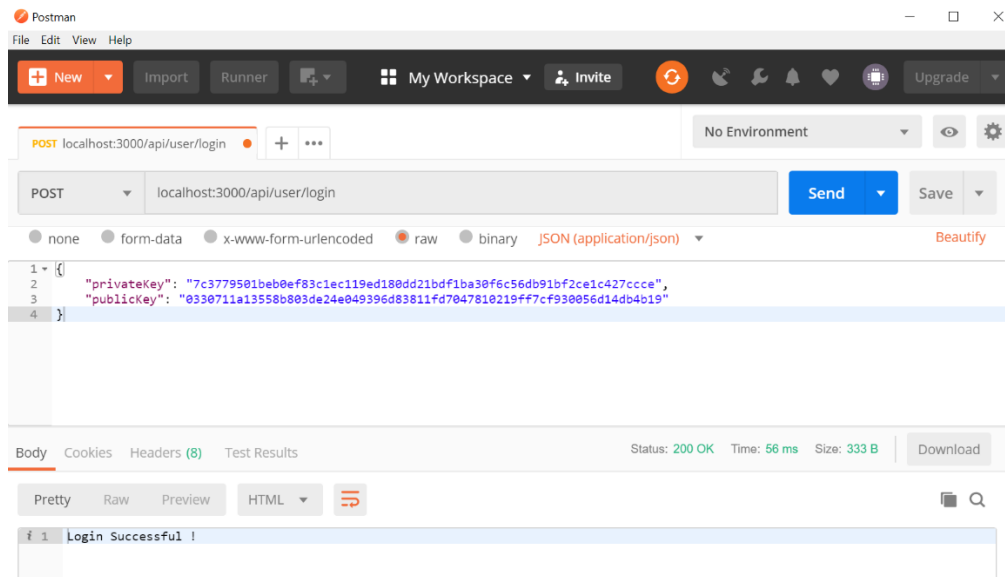


Figure 14. Miner Login API Sample Call and Response

This API is used when initializing the Miner Node. API access information given in Table 5. The Miner needs to login using his Wallet key pair. This private key will be used to sign all the newly generated blocks. This public key will be used as a payout address for all the block generation rewards.

5.2.3 Get Miner's Current Coin-Age API

Table 6. Get Miner's Coin-Age API

Request Type	GET
API End-Point	/api/user/getCoinAge
Sample URL	http://localhost:3000/api/user/getCoinAge
Input	<none>

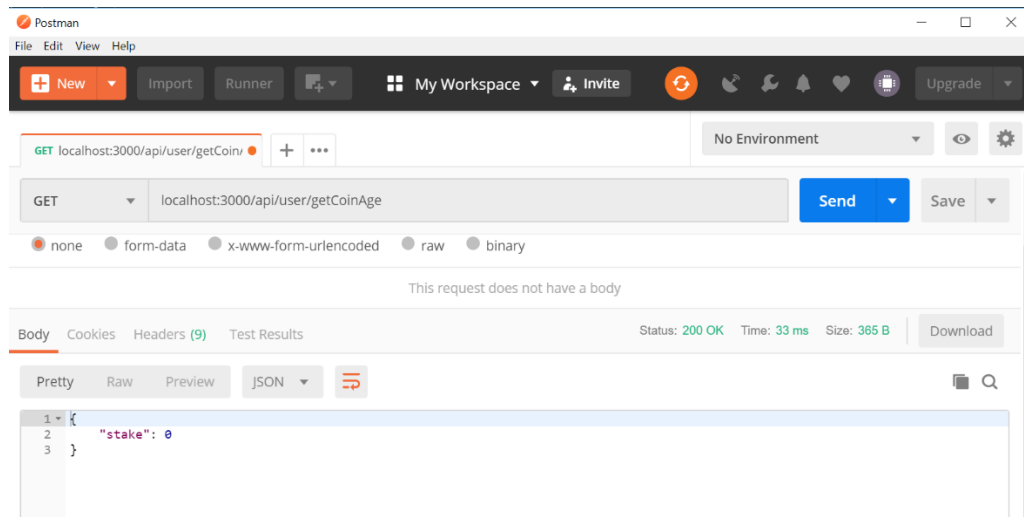


Figure 15. Get Miner's Coin-Age API Sample Call and Response

This API provides the Coin-Age held by the Logged-in Miner in the current block. Since we logged in with a newly generated Wallet key in the previous example, its stake is 0 since it has no balance. API access information given in Table 6.

5.2.4 Get Latest Block API

Table 7. Get Latest Block API

Request Type	GET
API End-Point	/api/block/latest
Sample URL	http://localhost:3000/api/block/latest
Input	<none>

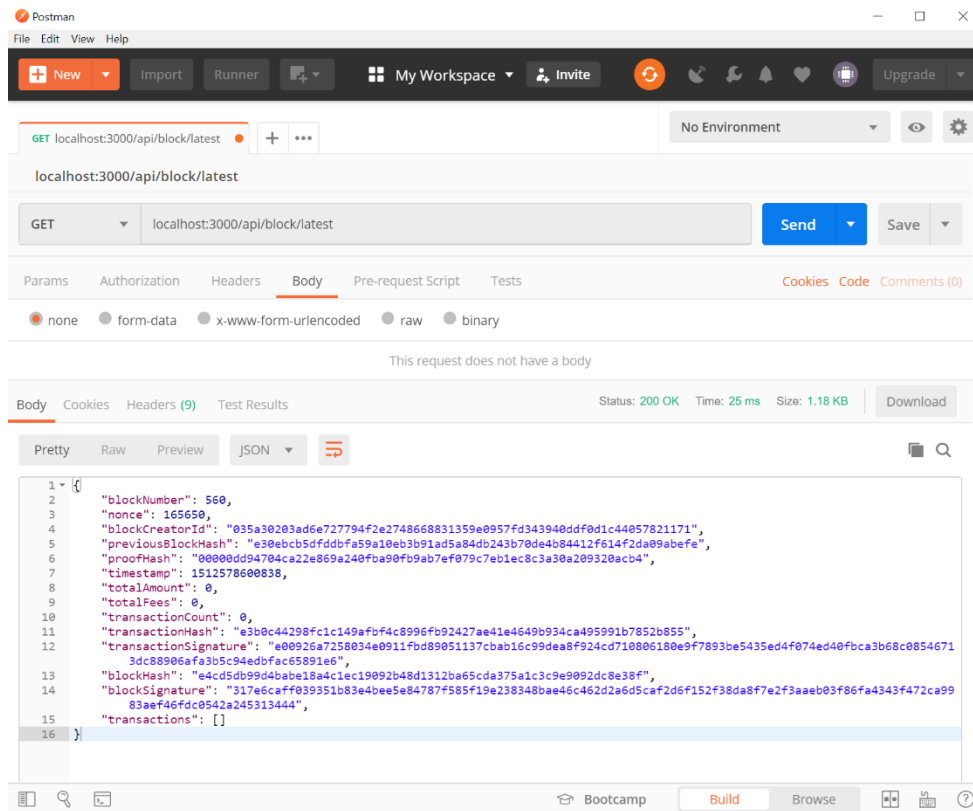


Figure 16. Get Latest Block API Sample Call and Response

The get latest block API can be used by End-Users to check the latest block in the blockchain. The output is a Block JSON object as discussed in the design phase. API access information given in Table 7.

5.2.5 Get User's Account Balance API

Table 8. Get User's Account Balance API

Request Type	GET
API End-Point	/api/balance/:userId
Sample URL	http://localhost:3000/api/balance/03409f79ae890a80dea6c4dfcc9b912b121553353570fee498af1c2d6b8c7e9f5
Input	The “:userId” in the URL must be replaced with a desired user's public key

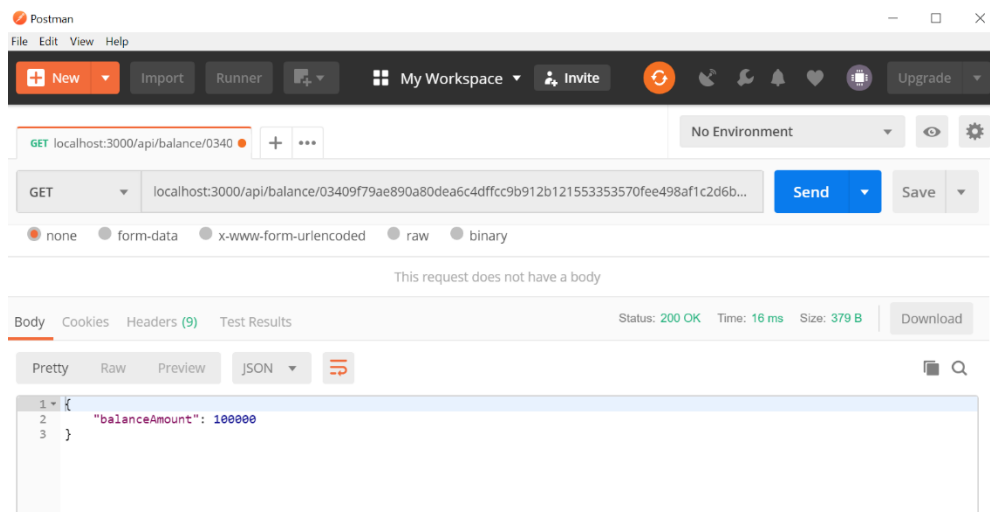


Figure 17. Get User's Account Balance API Sample Call and Response

This API can be used to check any user's current balance in the Ecash blockchain. The wallet address used in this example is one of the Wallets used while initializing the Genesis block. The genesis block details and pre-existing Wallet keys used for testing are available in the “extra” folder in the projects root directory. API access information given in Table 8.

5.2.6 Create New Transaction API

Table 9. Create New Transaction API

Request Type	GET
API End-Point	/api/transaction/create
Sample URL	http://localhost:3000/api/transaction/create
Input	The coin sender needs to provide transaction amount, fees, sender's private key, sender's and receiver's public keys and a deadline until which this unconfirmed transaction will remain valid

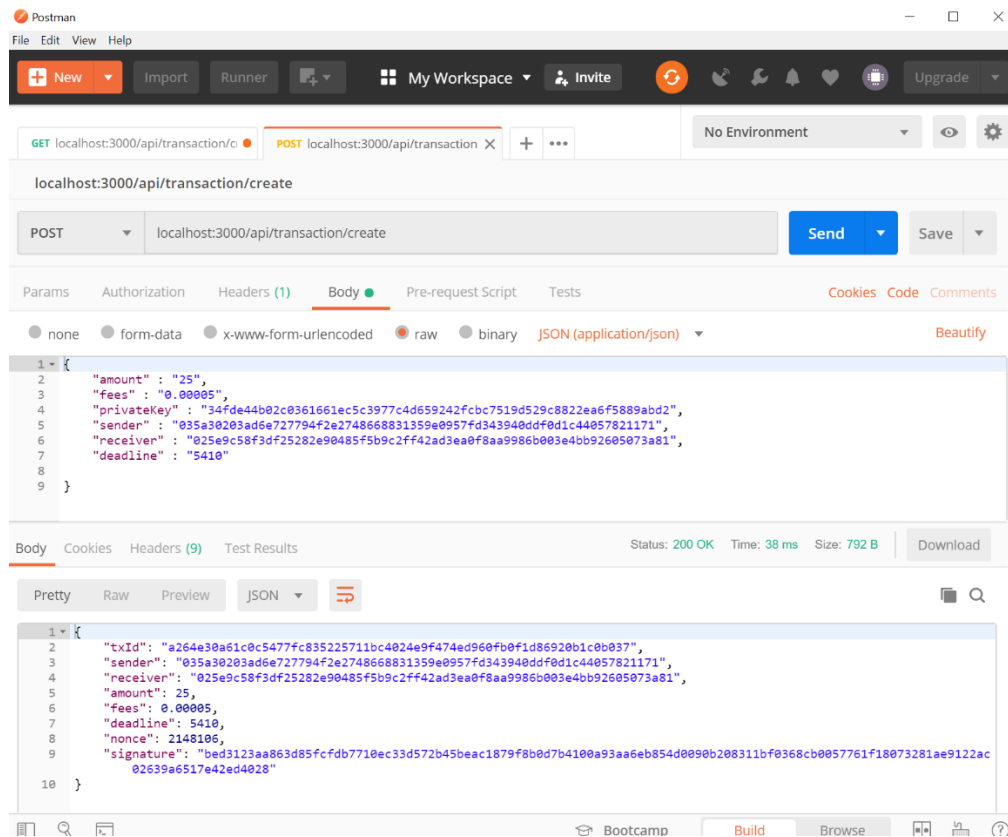


Figure 18. Create New Transaction API Sample Call and Response

End-Users can use this API to send coins to other users without any local processing requirements. The server (Wallet Node) generates the nonce,

signature for the transaction and generates a transaction id “txId”. This txId can be used to find if a particular unconfirmed transaction already exists in the server’s list of unconfirmed transactions.

However, since there may be security concerns for users sharing their Private keys, we can replace this step with generating the nonce and signature in the End-Users local machine and directly broadcasting the transaction object or making an API call to a Wallet Node to broadcast it for us. API access information given in Table 9.

5.2.7 Check Unconfirmed Transaction API

Table 2. Check Unconfirmed Transaction API

Request Type	GET
API End-Point	/api/transaction/unconfirmed/:transactionId
Sample URL	http://localhost:3000/api/transaction/unconfirmed/a264e30a61c0c5477fc835225711bc4024e9f474ed960fb0f1d86920b1c0b037
Input	The “:transactionId” in the URL needs to be replaced with a valid txId.

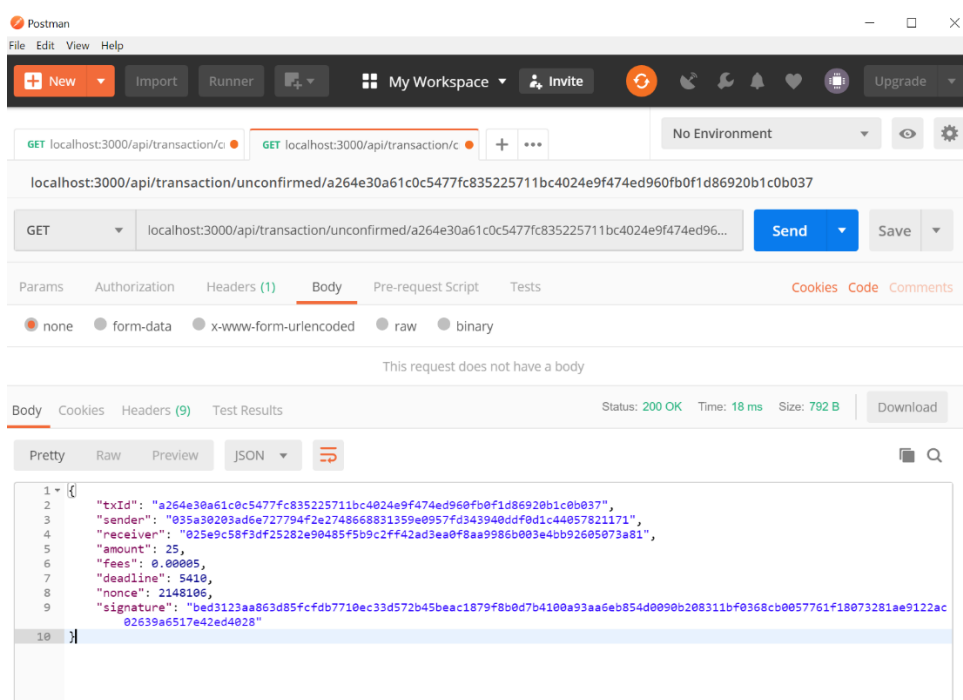


Figure 19. Check Unconfirmed Transactions API Sample Call and Response

I have passed the txId generated in the previous create transaction API call.

As we can see, our local Node now contains the transaction as an unconfirmed transaction. The Node is also responsible for broadcasting this

new transaction to all other nodes in its network and removing it from the unconfirmed transactions list once it has been added to the blockchain. API access information given in Table 10.

5.2.8 Non-API Background Activity

The Miner Nodes also perform a lot of other activities in the background which are not covered in the previously discussed API end-points. For example, the Miner Node continuously tries to generate a new block, send and receive unconfirmed transaction broadcasts, send and receive new block broadcasts, verify and validate all the incoming blocks and unconfirmed transactions, check Coin-Age of the current block head and manage forks in the blockchain when they occur. I will explain all of these in further detail in the Code Implementation section.

5.3 Code Implementation

In this section I will explain the basic code structure of the application and implementation of some important functions in detail. I will also explain the configuration files and explain which changes are permitted. All the code is written using JavaScript. Some parts of the code may be specific to Node.js implementation but should be convertible to a frontend executable JavaScript version with minimal modifications.

5.3.1 Folder Structure

We will first start with understanding the folder structure of the Miner and Wallet Nodes. Both the Nodes run over the same set of files and only differ in the instantiation process. The Miner Node starts the new Block Mining process while the Wallet does not.

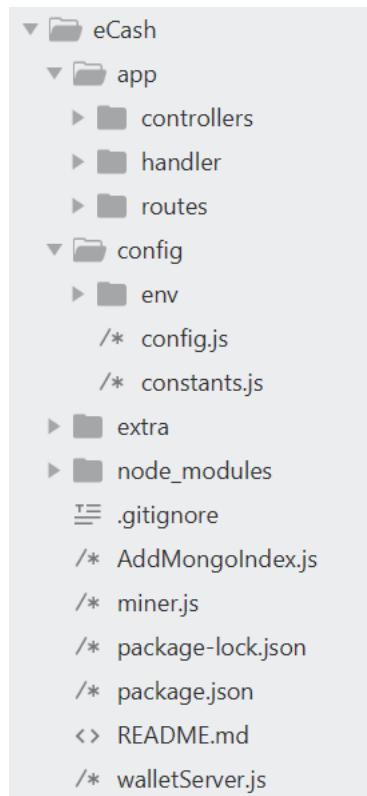


Figure 20. Folder Structure

1. The app folder contains all the application logic and API definitions.
The controllers, handler and routes are placed inside the app folder.
2. The controllers folder contains all the logical functions for the API calls and block generation.

3. The handler folder contains the functions used to access the MongoDB and Redis instances. It also contains the error handler that defines all the error codes.
4. The routes folder defines all the API routes and calls the corresponding functional implementation specified within the controller functions.
5. The config folder contains all the configuration files and constants in the code. Some of these values can be changed and will be explained in the next section.
6. The extra folder contains the Genesis Block – which means the first block of the blockchain – and some sample wallet key pairs used within the Genesis block.
7. node-modules is an auto-generated folder containing all the libraries required by the project. This folder is generated when we run the “npm install” command.
8. The rest of the files in the root directory contain the package.json which imports all required libraries. The addMongoIndex file adds indexes to MongoDB for unique block number validation and to speed-up the MongoDB query performance. Finally, it also contains the Wallet Node and Miner Nodes start-up files. Each file will be explained in detail in the next sections.

5.3.2 Package.json – Library usage

The package.json file contains all the library imports required by the Ecash application. All the libraries specified inside the package.json file will be installed automatically when the user runs “npm install”. The package file can also specify the version number for each library, to avoid any future changes to the library from breaking any existing code in the Ecash application.

```
{
  "name": "eCash",
  "version": "1.0.0",
  "description": "eCash coin wallet and miner",
  "dependencies": {
    "async": "^2.5.0",
    "express": "^4.16.2",
    "glob": "^7.1.2",
    "lodash": "^4.17.4",
    "mongodb": "^2.2.33",
    "redis": "^2.8.0",
    "rootpath": "^0.1.2",
    "secp256k1": "^3.3.0",
    "socket.io": "^2.0.4",
    "socket.io-client": "^2.0.4"
  },
  "author": "Rijul Luman"
}
```

Figure 21. package.json file

I'll now give some details about each library used in Ecash.

1. Async: Since Node.js is an asynchronous language, async can be used to better manage operations being performed asynchronously

2. Express: Express provides the RESTful API end-points and routing functionality to accept requests from the frontend
3. Glob: Used to bulk read the configuration and environment files when the we execute the application
4. Lodash: A modern JavaScript utility library delivering modularity, performance & extras. It provides some basic sorting, object and array handling operations useful throughout any Node.js project
5. Mongodb: The official Node.js driver provided by MongoDB to access and modify data and perform queries on the database
6. Redis: The official Node.js driver provided by Redis to add and remove key value pairs within the in-memory Redis datastore
7. Rootpath: Used to simplify file paths used within the project and access the files independent of the operating system using a common file path representation.
8. Secp256k1: Provides the Elliptic Curve " $y^2 = x^3 + 7$ " over a prime field. This library can generate the public-private elliptic curve key pairs and sign and verify signatures for these keys.
9. Socket.io and Socket.io-client: These libraries are used to open real-time communication ports with other Full Nodes on the network. Since each Node can open new outgoing connection requests and also needs to accept incoming connection requests, we need both the client-side and the server-side libraries from socket.io

5.3.3 Configuration Files

The configuration files and the environment specific configurations are present in the config folder. All the application-wide constants are stored in the constants.js file and all the configuration files are combined using the glob library inside the config.js file. Some values in constants.js are modifiable but these variables can only be changed before the blockchain has been initialized. After the initial application has been deployed, only the values in the configuration files are modifiable.

```

"SOCKET_BROADCAST_BLOCK"      : "blockBroadcast",
"SOCKET_BROADCAST_TRANSACTION" : "transactionBroadcast",
"SOCKET_GET_LATEST_BLOCK_HASHES" : "latestBlockHashes",
"SOCKET_GET_LATEST_BLOCK_REPLY" : "latestBlockHashesReply",
"FORK"                        : "fork",
"UPDATE"                     : "update",
"RESET"                      : "reset",
"MY_HASHES"                  : "myHashes",
"YOUR_UPDATE_STATUS"         : "updateStatus",
"NEXT_BLOCKS"                : "myBlocks",
"LAST_BLOCK_NUMBER"          : "lastBlock",
"NETWORK_BLOCK_SHARE_LIMIT"  : 20,

"PRIVATE_KEY_LENGTH"         : 32,
"VALID_HEX_PRIVATE_KEY_LENGTH" : 64,      // this.PRIVATE_KEY_LENGTH * 2,
"VALID_HEX_PUBLIC_KEY_LENGTH" : 66,
"SUM_DECIMAL_CORRECTION"     : 10000000,
"MINIMUM_TRANSACTION_NONCE"   : 1000000,
"BLOCK_MAX_TRANSACTIONS_COUNT" : 512,
"GENESIS_BLOCK_PREV_HASH"     : "0123456789abcdeffedcba98765432100123456789abcdeffedcba9876543210",
"GENESIS_BLOCK_TARGET"        : "03ffffff", // 03ffffff is approximately 1 min
"DIFFICULTY_CHANGE_EVERY_BLOCKS" : 1000,
"TRANSACTION_DEADLINE_OFFSET" : 100,      // Number of blocks before expiring
"MAX_STAKEABLE_BLOCKS"        : 100,
"MIN_HOLD_FOR_STAKE_BLOCKS"    : 20,

"AVERAGE_BLOCK_TIME_MS"      : 60000,    // 60000 * 1,      // 1 min
"CURRENT_BLOCK_REDIS_TTL"      : 200,      // Ideally a little higher than the average block time
"BLOCK_COIN_AGE_REDIS_TTL"     : 200,      // Ideally a little higher than the average block time
"BLOCKCHAIN_UPDATE_HOLD_TTL_MULTIPLIER" : 0.5, // Should be higher than 2
"PROOF_HASH_AVG_BLOCK_TIME_MULTIPLIER" : 2,
"UNCONFIRMED_TRANSACTION_TTL_SECONDS_PER_BLOCK" : 3600, // Seconds per block difference
"UPDATE_REQUEST_BLOCK_HASH_COUNT" : 10,
"CREATE_BLOCK_UPDATE_TIMEOUT"   : 3000,    // in ms

"redisPath" : {
  "unconfirmedTransaction" : "UT.",
  "sortedUnconfirmedTransaction" : "SUT",
  "currentBlock" : "CB",
  "coinAge" : "AGE.",
  "setUpdateSocketId" : "UID",
  "blockchainUpdateInProgress" : "BUP",
  "userKeys" : "USER"
}

```

Figure 22. Constants defined within the application

Some of the values in Figure 22 can be changed to further optimize the system or change the system entirely. I have explained all the values that can be changed in Tables 11a, 11b, 11c and 11d below and the restrictions for such changes. The values used in the configuration file above were found to be the best when testing the network over three inter-connected Full Nodes.

Table 11a. Modifiable configuration variables

Variable Name	Possible changes & Restrictions to such changes
NETWORK_BLOCK_SHARE _LIMIT	This limits the number of blocks transferred to an updating Node. This restriction is important to maintain the blockchain update process. HTTP requests are generally capable of handling about 16mb data in each request. This number maintains the data body below 16mb to keep the network operations stable.
MINIMUM_TRANSACTION_ NONCE	This is the minimum number required as the nonce for a transaction. A larger number keeps the nonce string representation larger, meaning the salt for each transaction is larger than this number.
BLOCK_MAX_TRANSACTION_ COUNT	The Maximum transactions confirmable within a single block. This value along with AVERAGE_BLOCK_TIME_MS determines the transaction speed of blockchain. Since the blocks are generated at a fixed pace, the maximum number of transactions within each block defines the theoretical maximum transaction processing speed for Ecash

Table 3b. Modifiable configuration variables

Variable Name	Possible changes & Restrictions to such changes
GENESIS_BLOCK_TARGET	<p>The target difficulty value for the Genesis block. This sets the mining difficulty the blockchain starts with when it is initialized. The first 2 digits specify the number of zero hexadecimal digits required by the proofHash and the next 6 hexadecimal digits specify the minimum value for the next 6 hexadecimal digits in the proofHash. After some testing the value 03ffffff takes about 1-minute to mine a block on a single 8-core system. Since the target value changes automatically as the blockchain progresses, changes to this value will only affect the first DIFFICULTY_CHANGE_EVERY_BLOCKS number of blocks. This value works in conjunction with AVERAGE_BLOCK_TIME_MS. Both these variables together define the average speed for block generation maintained by the Ecash consensus algorithm.</p>

Table 4c. Modifiable configuration variables

Variable Name	Possible changes & Restrictions to such changes
AVERAGE_BLOCK_TIME_ MS	This is the expected average block generation time to be maintained for a every block. This value is in milliseconds and is set to 60000ms that is 1-minute.
DIFFICULTY_CHANGE_EVE RY_BLOCKS	This value defines when the target value for the blockchain difficulty needs to be recalculated. I have set it to 1000. Since the default setting take 1-minute on average for the block generation, 1000 blocks are generated every 16-17 hours hence the network will re-calibrate its difficulty at least once every day.
MAX_STAKEABLE_BLOCK S	This is the maximum number of blocks to be considered for the stake calculations. The stake calculation starts from the first block to be considered for this user and ends at first block number + MAX_STAKEABLE_BLOCKS. This number limits the maximum possible Coin-Age a single user can obtain. Increasing this number will increase the max possible stake for each individual Miners.

Table 5d. Modifiable configuration variables

Variable Name	Possible changes & Restrictions to such changes
MIN_HOLD_FOR_STAKE_BLOCKS	This value defines the minimum number of blocks a Miner needs to hold their coins before they become stake-able. This is done to stop malicious users from transferring all their coins to a new Wallet address hoping to reset their Coin-Age to its possible highest value.

The ports to be used by this application can be changed in the env folder, inside the development.js file as shown in Figure 23 below. The env folder also contains a production.json file, the user can use a different environment file with production Port and IP values, if required.

```

{
  "mongo_url": "mongodb://localhost:27017/eCash",
  "mongo_coll_block" : "blocks",
  "mongo_coll_balance" : "balances",
  "mongo_coll_target" : "targets",

  "redis" : {
    "MA" : {
      "host" : "127.0.0.1",
      "port" : 6379
    },
    "SL" : {
      "host" : "127.0.0.1",
      "port" : 6379
    }
  },

  "default_broadcast_sockets" : [],
  "miner_broadcast_sockets" : [],
  "express_port": 3000,
  "socket_io_port" : 3001,
  "miner_io_port" : 3002
}

```

Figure 23. Ports and Network Nodes Configuration File

The collection names and the MongoDB server location and Redis server location can be modified here. These services may be local to the node, as shown in Figure 23, or maybe external IP addresses. The user can add comma separated IP addresses with port numbers in “xxx.xxx.xxx.xxx:port” format for the “default_broadcast_sockets” and the “miner_broadcast_sockets”.

The “default_broadcast_sockets” are used to update our local blockchain after being offline for any amount of time. The “miner_broadcast_sockets”

are the network Nodes to whom we broadcast new blocks and unconfirmed transactions.

5.3.4 Wallet Key Generation

The Wallet key generation process is a straight forward way to generate Public-Private Elliptic Curve key pairs using the Secp256k1 library. It contains the standard implementation for the elliptic curve $y^2 = x^3 + 7$ over a prime field. Since this elliptic curve is fairly new, the internal Node.js crypto library does not have this curve's implementation.

```
exports.generate = function(req, res, next) {
  var privateKey;
  do {
    privateKey = crypto.randomBytes(Constants.PRIVATE_KEY_LENGTH);
  } while (!secp256k1.privateKeyVerify(privateKey))

  var publicKey = secp256k1.publicKeyCreate(privateKey, true);

  var jsonResp = {
    privateKey : CommonFunctions.bufferToHexString(privateKey),
    publicKey  : CommonFunctions.bufferToHexString(publicKey)
  };

  res.jsonp(jsonResp);
};
```

Figure 24. Wallet key generation using the secp256k1 library

The generate function in Figure 24 returns a JSON representation of the key as the HTTP response for the key generation API call.

The library also provides functions to use private keys to sign a message and use a public key to verify a signature as shown in Figure 25.

```

generateSignature : function(sha256Hash, privateKey){
  return CommonFunctions.bufferToHexString(secp256k1.sign(CommonFunctions.hexStringToBuffer(sha256Hash), privateKey).signature);
},

verifySignature : function(sha256Hash, publicKey, signature){
  return secp256k1.verify(CommonFunctions.hexStringToBuffer(sha256Hash), CommonFunctions.hexStringToBuffer(signature), CommonFunctions.hexStringToBuffer(publicKey));
},

verifyWalletKeyPair : function(privateKey, publicKey){
  var sampleMessage = Constants.GENESIS_BLOCK_PREV_HASH;
  privateKey = CommonFunctions.hexStringToBuffer(privateKey);
  return CommonFunctions.verifySignature(sampleMessage, publicKey, CommonFunctions.generateSignature(sampleMessage, privateKey));
}

```

Figure 25. Supporting functions for Signature generation and verification

5.3.5 Unconfirmed Transaction Generation

The Wallet Node is responsible for creating Unconfirmed Transactions from the generate transactions API. It generates a valid nonce and hash for the transaction and also signs the final unconfirmed transaction object for the user.

```

exports.create = function (req, res, next) {
  var transaction = {
    txId      : "", // Unique Hash for this transaction
    sender    : "",
    receiver  : "",
    amount    : 0,
    fees      : 0,
    deadline  : 0,
    nonce     : 0,
    signature : ""
  };
  if(!req.body){
    return ErrorHandler.getErrorJSONData({'code':5, 'res':res});
  }
  transaction.amount = validateCoinValue(req.body.amount);
  if(!transaction.amount){
    return ErrorHandler.getErrorJSONData({'code':6, 'res':res});
  }
  transaction.fees = validateCoinValue(req.body.fees);
  if(!transaction.fees){
    return ErrorHandler.getErrorJSONData({'code':7, 'res':res});
  }
  if(!req.body.sender || !CommonFunctions.validatePublicKeyHexString(req.body.sender)){
    return ErrorHandler.getErrorJSONData({'code':8, 'res':res});
  }
  transaction.sender = req.body.sender.toLowerCase();
  if(!req.body.receiver || !CommonFunctions.validatePublicKeyHexString(req.body.receiver)){
    return ErrorHandler.getErrorJSONData({'code':9, 'res':res});
  }
  transaction.receiver = req.body.receiver.toLowerCase();
  if(transaction.sender == transaction.receiver){
    return ErrorHandler.getErrorJSONData({'code':16, 'res':res});
  }
  if(!req.body.privateKey || !CommonFunctions.validatePrivateKeyHexString(req.body.privateKey)){
    return ErrorHandler.getErrorJSONData({'code':10, 'res':res});
  }
  var privateKey = CommonFunctions.hexStringToBuffer(req.body.privateKey.toLowerCase());

  async.waterfall([
    function defaultDeadline(cb){
      if(!req.body.deadline){
        RedisHandler.getCurrentBlock(function(err, deadline){
          if(err){
            return ErrorHandler.getErrorJSONData({'code':3, 'res':res});
          }
          else{
            transaction.deadline = deadline + Constants.TRANSACTION_DEADLINE_OFFSET;
          }
          cb();
        });
      }
      else if(isNaN(parseInt(req.body.deadline))){
        return ErrorHandler.getErrorJSONData({'code':11, 'res':res});
      }
      else{
        transaction.deadline = parseInt(req.body.deadline);
        cb();
      }
    },
    function generateNonce(cb){
      CommonFunctions.generateTransactionNonce(function(err, nonce){
        if(err){
          return ErrorHandler.getErrorJSONData({'code':12, 'res':res});
        }
        transaction.nonce = nonce;
        cb();
      });
    },
    function generateTransactionId(cb){
      transaction.txId = CommonFunctions.generateTransactionHash(transaction);
      cb();
    },
    function generateSignature(cb){
      transaction.signature = CommonFunctions.generateSignature(transaction.txId, privateKey);
      cb();
    }
  ], function(){
    addUnconfirmedTransaction(transaction);
    broadcast(transaction);
    res.status(200).jsonp(transaction);
  });
};

```

Figure 26. Unconfirmed Transaction Generation Function

```
generateTransactionHash : function(transaction){  
  var hash = crypto.createHash('sha256');  
  hash.update("" + transaction.nonce );  
  hash.update(transaction.sender );  
  hash.update(transaction.receiver );  
  hash.update("" + transaction.amount );  
  hash.update("" + transaction.fees );  
  hash.update("" + transaction.deadline );  
  return hash.digest('hex');  
},
```

Figure 27. Hashing Function for transaction generation

For generating the SHA256 hash value for the transaction, I have used Node.js's internal crypto library. The generateTransactionHash function takes in all transaction object's values as strings and returns the hash value for the any given transaction as a hexadecimal string.

5.3.6 Verify and Store Unconfirmed Transaction Broadcasts

The Node will receive broadcasts of new unconfirmed transactions all the time once users start making payments over the Ecash network. These unconfirmed transactions need to be verified and added into the list of unconfirmed transactions, so that they can be confirmed in the future blocks.

```
exports.acceptBroadcastTransaction = function(transaction){
  console.log("Incoming BroadcastTransaction : ", transaction);
  addUnconfirmedTransaction(transaction);
};

/**
 * Validate and add transaction to Redis
 */
var addUnconfirmedTransaction = function(transaction){
  validateAndParseTransaction(transaction, function(isValid, validatedTransaction){
    if(isValid){
      RedisHandler.addUnconfirmedTransaction(validatedTransaction, function (err, reply) {
        // Transaction Added
      });
    }
    else{
      console.log("Invalid Transaction could not be added");
    }
  });
};

var broadcast = function (transaction) {
  validateAndParseTransaction(transaction, function(isValid, validatedTransaction){
    if(isValid){
      BroadcastMaster.sockets.emit(Constants.SOCKET_BROADCAST_TRANSACTION, validatedTransaction);
      OutgoingSockets.forEach(function(socket){
        socket.emit(Constants.SOCKET_BROADCAST_TRANSACTION, validatedTransaction);
      });
    }
  });
};
```

Figure 28. Adding and Broadcasting Unconfirmed Transactions

Ecash uses Redis to store all unconfirmed transactions. Any node losing power will also lose all the unconfirmed transactions it holds. It can obtain this list from other nodes when it restarts.


```

var validateAndParseTransaction = function (transactionInput, callback) {
  if(!transactionInput || typeof(transactionInput) !== "object" || Object.keys(transactionInput).length < 8){
    return callback(false, null);
  }

  var transaction = {};

  try{
    transaction = {
      txId      : transactionInput.txId.toLowerCase(),
      sender    : transactionInput.sender.toLowerCase(),
      receiver   : transactionInput.receiver.toLowerCase(),
      amount    : validateCoinValue(transactionInput.amount),
      fees      : validateCoinValue(transactionInput.fees),
      deadline  : parseInt(transactionInput.deadline),
      nonce     : parseInt(transactionInput.nonce),
      signature  : transactionInput.signature.toLowerCase()
    };
  }
  catch(e){
    return callback(false, null);
  }

  if(
    !transaction.amount
    || !transaction.fees
    || !transaction.sender
    || !CommonFunctions.validatePublicKeyHexString(transaction.sender)
    || !transaction.receiver
    || !CommonFunctions.validatePublicKeyHexString(transaction.receiver)
    || !transaction.nonce
    || isNaN(parseInt(transaction.nonce))
    || parseInt(transaction.nonce) < Constants.MINIMUM_TRANSACTION_NONCE
    || transaction.sender !== transaction.receiver
    || CommonFunctions.generateTransactionHash(transaction) !== transaction.txId
    || !CommonFunctions.verifySignature(transaction.txId, transaction.sender, transaction.signature)
  ){
    return callback(false, null);
  }

  callback(true, transaction);
};

```

Figure 29. Validating Unconfirmed Transactions

It is important to validate all incoming unconfirmed transactions from the network Nodes since these transactions may simply be invalid or may be modified by an attacker. Validating unconfirmed transactions before adding them into the Nodes memory guarantees any unconfirmed transaction picked up for a new block will always be valid – it's sender may not have the required balance, but the transaction object's format will be valid.

```

addUnconfirmedTransaction: function(transaction, callback) {
  async.waterfall([
    function getCurrentBlockNumber(cb){
      RedisHandler.getCurrentBlock(cb);
    },
    function setTransaction(currentBlock, cb){
      // Set TTL to block difference * block time
      var blockDifference = transaction.deadline - currentBlock;
      if(blockDifference < 0){
        return callback(null, true); // Expired transaction, need not be added to unconfirmed Transactions
      }
      var ttl = blockDifference * Constants.UNCONFIRMED_TRANSACTION_TTL_SECONDS_PER_BLOCK;
      RedisStoreMA.setex(redisPath.unconfirmedTransaction + transaction.txId, ttl, JSON.stringify(transaction), function(err, reply){
        cb(err);
      });
    },
    function sortTransactionByFee(cb){
      RedisStoreMA.zadd([redisPath.sortedUnconfirmedTransaction, transaction.fees * 10000000, transaction.txId], function(err, reply){
        cb(err);
      });
    },
  ],
  function(errs, result){
    if(errs && errs.length){
      callback(true, null);
    }
    else{
      callback(null, true);
    }
  });
},

```

Figure 30. Storing Unconfirmed Transactions into Redis

Once the unconfirmed transaction has been validated, we store it into Redis using the function defined in Figure 30.

5.3.7 New Block Generation

Most of the computational complexity occurs during a new block creation process – known as Mining. The Miner Node reads the first `BLOCK_MAX_TRANSACTIONS_COUNT` number of highest fee-paying unconfirmed transactions from its unconfirmed transaction store. These transactions are first validated to figure out if the sender of these transactions actually has sufficient balance to perform this action and combined into a single transaction array. A cumulative transaction hash is generated for this array and signed by the Miner's Private key. The Miner then proceeds to find a valid proofHash for this block. Once the Miner finds

a solution it signs the block and broadcasts the block over the network. The network Nodes then accept the new block and verify if each of the transactions in the block is valid and each transaction sender had sufficient balance to make those transactions. This process is done before accepting any new block, the Miner is always sure about the account balances, as the Miner checks the balances at each step before storing blocks into their local copy of the blockchain. Once the incoming new block has been accepted, the Miners can start building on top of this latest block by using its blockHash as the previousBlockHash for the next block. This process keeps going on and on until the Miner Node is shutdown. Miner Nodes and Wallet Nodes update their local blockchain when they restart to ensure they are working on the most up-to-date version of the blockchain.

```

var createBlockLocal = function(user, callback) {
  var block = {
    blockNumber      : 0,
    nonce            : 0,
    blockCreatorId    : "",
    previousBlockHash : "",
    proofHash         : "",
    timestamp         : 0,
    totalAmount       : 0,
    totalFees         : 0,
    transactionCount   : 0,
    transactionHash    : "",
    transactionSignature : "",
    blockHash         : "",
    blockSignature     : "",
    transactions       : []
  };

  var zaddClear = [];

  async.waterfall([
    function getBlockCreatorDetails(cb){
      block.blockCreatorId = user.publicKey;
      cb();
    },

    function getTransactionArray(cb){
      makeTransactionArray(Constants.BLOCK_MAX_TRANSACTIONS_COUNT, function(err, ids, transactions){
        block.transactions = transactions;
        block.transactionCount = transactions.length;
        zaddClear = ids; // Transaction ids to remove after creation of block
        cb(err);
      });
    },

    function calculateTotalCoins(cb){
      block.transactions.forEach(function(transaction){
        block.totalFees += transaction.fees * Constants.SUM_DECIMAL_CORRECTION;
        block.totalAmount += transaction.amount * Constants.SUM_DECIMAL_CORRECTION;
      });
      block.totalFees = block.totalFees / Constants.SUM_DECIMAL_CORRECTION;
      block.totalAmount = block.totalAmount / Constants.SUM_DECIMAL_CORRECTION;
      cb();
    },

    function updateBlockChain(cb){
      // Also delete unconfirmed transactions present in block from memory
      MongoHandler.updateBlockchain();
      setTimeout(function(){ cb(); }, Constants.CREATE_BLOCK_UPDATE_TIMEOUT);
    },

    function getPreviousBlock(cb){
      MongoHandler.getCurrentBlock(function(errs, previousBlock){
        block.blockNumber = previousBlock.blockNumber + 1;
        block.previousBlockHash = previousBlock.blockHash;
        cb();
      });
    },

    function generateHashesAndSignatures(cb){
      block.transactionHash = CommonFunctions.generateTransactionArrayHash(block.transactions);
      block.transactionSignature = CommonFunctions.generateSignature(block.transactionHash, user.privateKey);

      MongoHandler.getTargetForBlock(block.blockNumber, function(err, target){
        console.time("BlockGenerationTime");
        var nonceAndHash = CommonFunctions.generateProofHashAndNonce(target, block);
        block.nonce = nonceAndHash.nonce;
        block.proofHash = nonceAndHash.hash;
        block.timestamp = new Date().getTime();
        block.blockHash = CommonFunctions.generateBlockHash(block);
        block.blockSignature = CommonFunctions.generateSignature(block.blockHash, user.privateKey);
        console.timeEnd("BlockGenerationTime");
        cb();
      });
    },

    function validateGeneratedBlock(cb){
      validateAndParseBlock(block, function(isValid, parsedBlock){
        if(!isValid){
          console.log("Invalid Block Generated !!", JSON.stringify(block, null, 2));
          block = parsedBlock;
          cb(!isValid);
        }
      });
    },

    function addBlockToDb(cb){
      acceptBroadcastBlock(block); // We calculate stake, incase the DB has received the next block from the network
      cb();
    },

    function broadcastGeneratedBlock(cb){
      block._id = undefined; // Remove MongoDB _id from record before broadcast
      broadcastBlock(block);
      cb();
    },

    function removeTransactionsFromMemory(cb){
      RedisHandler.removeTransactionsFromZList(zaddClear);
      RedisHandler.removeUnconfirmedTransactions(block.transactions);
      RedisHandler.clearCurrentBlock();
      cb();
    }
  ], function(errs, result){
    console.log("Block Done !");
    callback(null, block);
  });
};

```

Figure 31. New Block Generation Function

```

var makeTransactionArray = function(count, callback){
  RedisHandler.getMaxFeeTransactionIds(count, function(err, ids){
    if(ids.length == 0){
      return callback(null, [], []);
    }
    RedisHandler.getTransactionArray(ids, function(err, txArr){
      validateAccountBalances(txArr, function(err, validTxArr, invalidTxArr){
        invalidTxArr.forEach(function(tx){
          ids.splice(ids.indexOf(tx.txId), 1);
        });
        removeTransactionsAlreadyInBlockChain(validTxArr, function(err, finalTxArr){
          if(finalTxArr.length < count && ids.length == count){
            makeTransactionArray(count - finalTxArr.length, function(err, recId, recArr){
              callback(null, ids.concat(recId), finalTxArr.concat(recArr));
            });
          }
          else{
            callback(null, ids, finalTxArr);
          }
        });
      });
    });
  });
};

```

Figure 32. Unconfirmed Transactions to Confirmed Transaction Array Function

The `makeTransactionArray` function in Figure 32 calls itself recursively until it finds `BLOCK_MAX_TRANSACTIONS_COUNT` number of transactions with valid account balances or until it is out of valid unconfirmed transactions. During this array creation we also check if the unconfirmed transaction has already been confirmed by some other Node in one of the previous blocks, any transaction that was already confirmed will be replaced with other valid unconfirmed transactions. All the unconfirmed transactions that are invalid or have already been confirmed in a previous block will be removed from Redis using the `removeTransactionAlreadyInBlockChain` function as shown in Figure 34.

```
var validateAccountBalances = function(transactions, callback){
  var validTx = [];
  var invalidTx = [];

  async.each(transactions, function(transaction, cb){
    MongoHandler.calculateAccountBalance(transaction.sender, function(err, balance){
      if(balance < (transaction.amount + transaction.fees)){
        invalidTx.push(transaction);
      }
      else{
        validTx.push(transaction);
      }
      cb();
    });
  }, function(errs, results){
    callback(null, validTx, invalidTx);
  });
};
```

Figure 33. Account Balance Validation for Transactions Function

The `validateAccountBalances` function in Figure 33 checks if the transaction sender actually owns the amount of coins specified in the transaction plus the fee promised for processing the transaction. All transactions that do not have the required balance can be discarded or kept in the unconfirmed transactions list hoping to have sufficient balance during one of the next block generation cycle.

```

var removeTransactionsAlreadyInBlockChain = function(transactions, callback){
  var validTransactions = [];
  async.each(transactions, function(transaction, cb){
    BlockCollection.find({"transactions.txId" : transaction.txId}, {_id : 1}).limit(1).toArray(function(err, docs){
      if(docs && docs.length){
        RedisHandler.removeTransactionsFromZlist([transaction.txId]);
        RedisHandler.removeUnconfirmedTransactions([transaction]);
      }
      else{
        validTransactions.push(transaction);
      }
      cb();
    });
  }, function(errs, result){
    callback(null, validTransactions);
  });
};

```

Figure 34. Duplicate Transaction Removal Function

It is important to not have the same transaction multiple times in the blockchain, as it would mean the sender sent the same amount 2 times. Hence, we check and remove the transaction from the unconfirmed transactions list, if it has already been processed.

```

generateBlockHash : function(block){
  var hash = crypto.createHash('sha256');

  hash.update(""+block.blockNumber);
  hash.update(""+block.nonce );
  hash.update(block.proofHash);
  hash.update(""+block.timestamp);
  hash.update(block.blockCreatorId );
  hash.update(block.previousBlockHash );
  hash.update(""+block.totalAmount );
  hash.update(""+block.totalFees );
  block.transactions.forEach(function(transaction){
    hash.update(transaction.txId );
    hash.update("" + transaction.nonce );
    hash.update(transaction.sender );
    hash.update(transaction.receiver );
    hash.update("" + transaction.amount );
    hash.update("" + transaction.fees );
    hash.update("" + transaction.deadline );
    hash.update("" + transaction.signature );
  });
  hash.update(""+block.transactionCount );
  hash.update(block.transactionHash );
  hash.update(block.transactionSignature);

  return hash.digest('hex');
},

```

Figure 35. Block Hash Generation Function

Once the block has been generated with a valid proofHash, it needs to be signed with the block creators private key. We generate the block hash using all the values inside the block in string representation, similar to the transaction hash process. The generated block hash value is signed using the block creator's private key in order to receive the block generation rewards by proving they were really the ones responsible for Mining this

block. The blockHash is used as the previousBlockHash in the next block, providing core security to the blockchain.

5.3.8 Broadcasting Blocks and managing Forks in the blockchain

Every Full Node needs to remain updated with the network's blockchain to provide valid API responses. Every successfully Mined block is broadcasted onto the network and every incoming block is verified and validated before being inserted into the blockchain. When a Node leaves and rejoins the network, it sends some information about its present blockchain to the nodes specified in the default_broadcast_sockets in the development.json file as seen previously in Figure 23. Other Nodes check their version of the blockchain and send appropriate blocks to help update the requesting Nodes blockchain.

```
sendDataToRandomNodeInNetwork : function(socketCommand, data){
  // Currently only outgoing nodes used for random connections
  var socket = OutgoingSockets[Math.floor(Math.random() * (OutgoingSockets.length))];
  while(socket == null && Object.keys(BroadcastMaster.sockets.connected).length){
    var sockets = Object.keys(BroadcastMaster.sockets.connected);
    var randomIndex = Math.floor(Math.random() * (sockets.length + 1));
    socket = BroadcastMaster.sockets.connected[sockets[randomIndex]];
  }

  if(socket){
    var socketId = socket.id ? socket.id : socket.io.opts.hostname;
    RedisStoreWA.expire(redisPath.blockchainUpdateInProgress, Constants.BLOCKCHAIN_UPDATE_HOLD_TTL_MULTIPLIER * Constants.AVERAGE_BLOCK_TIME_MS / 1000); // Extend update TTL
    RedisHandler.setUpdaterDetails(socketId, function(err, reply){ // Save random node name in redis, only update on reply from that node (For Security)
      socket.emit(socketCommand, data);
    });
  }
},
```

Figure 36. Generated Block Broadcast Function

```

exports.receiveLatestBlocks = function(responseData, responseSocket){
  async.parallel([
    function(cb){
      RedisHandler.isBlockchainUpdateInProgress(cb);
    },
    function(cb){
      RedisHandler.getUpdateDetails(cb);
    }
  ], function(errs, results){
    var socketId = responseSocket.id; responseSocket.id = responseSocket.io.opts.hostname;
    if(results[0] || results[1] == socketId){
      if(responseData[Constants.YOUR_UPDATE_STATUS] == Constants.UPDATE){
        MongoDBHandler.insertNetworkBlocks(responseData[Constants.NEXT_BLOCKS], function(){
          MongoDBHandler.getCurrentBlockNumber(function(err, blockNumber){
            if(blockNumber < parseInt(responseData[Constants.LAST_BLOCK_NUMBER])){
              MongoDBHandler.updateBlockchainFromBlock(blockNumber); // Recursive call till we reach the latest block
            } else{
              RedisHandler.resetBlockchainUpdateInProgress();
            }
          });
        });
      } else if(responseData[Constants.YOUR_UPDATE_STATUS] == Constants.FORK){
        var updateBlocks = [];
        MongoDBHandler.getCurrentBlockNumber(function(err, blockNumber){
          for(var i = 0; i < responseData[Constants.NEXT_BLOCKS].length; i++){
            if(responseData[Constants.NEXT_BLOCKS][i].blockNumber == blockNumber){
              updateBlocks.push(responseData[Constants.NEXT_BLOCKS][i]);
              responseData[Constants.NEXT_BLOCKS].splice(i, 1);
            }
          }
          MongoDBHandler.updateNetworkBlocks(updateBlocks, function(){
            MongoDBHandler.insertNetworkBlocks(responseData[Constants.NEXT_BLOCKS], function(){
              MongoDBHandler.getCurrentBlockNumber(function(err, blockNumber){
                if(blockNumber < parseInt(responseData[Constants.LAST_BLOCK_NUMBER])){
                  MongoDBHandler.updateBlockchainFromBlock(blockNumber); // Recursive call till we reach the latest block
                } else{
                  RedisHandler.resetBlockchainUpdateInProgress();
                }
              });
            });
          });
        });
      } else if(responseData[Constants.YOUR_UPDATE_STATUS] == Constants.RESET){
        var blockNumbers = [];
        var blockHashes = [];
        responseData[Constants.NEXT_BLOCKS].forEach(function(block){
          blockNumbers.push(block.blockNumber);
          blockHashes.push(block.blockHash);
        });
        BlockCollection.find({ $in: blockNumbers }, { $in: blockHashes }, { _id: 0, blockNumber: 1, blockHash: 1 }).sort({blockNumber: -1}).toArray(function(err, matchedBlocks){
          if(matchedBlocks.length == matchedBlocks[0].blockNumber){
            MongoDBHandler.updateBlockchainFromBlock(matchedBlocks[0].blockNumber); // Recursive call till we reach a forking point
          } else if(matchedBlocks.length == 0){
            // If we have checkpoints, checkpoint number will be passed from here
            MongoDBHandler.updateBlockchainFromBlock(0); // Since none of the blocks match
          } else{
            RedisHandler.resetBlockchainUpdateInProgress();
          }
        });
      } else{
        // Invalid case
        RedisHandler.resetBlockchainUpdateInProgress();
      }
    }
  });
});

```

Figure 37. Receive Broadcasted Blocks from Network Function

Figure 36 contains the function responsible for handling incoming blockchain update requests. Whenever a new Node is added to the network or an existing node returns to the network after being offline for some time, it needs to update its local blockchain to begin Mining for the appropriate block and to be able to provide accurate values for API requests. The outgoing update requests are implemented in receiveLatestBlocks function as shown in Figure 37.

5.3.9 Coin-Age Calculations during Block Processing

The Miner needs to calculate the users Coin-Age when accepting new blocks from the network to calculate this blocks priority. This calculation helps the Miner Node decide whether the incoming block should replace the existing latest block or be ignored.

The Coin-Age calculation sums the coins held for “MIN_HOLD_FOR_STAKE_BLOCKS” number of blocks up to “MAX_STAKEABLE_BLOCKS” times.

```

calculateCoinAge : function(userId, suppliedBlockNumber, callback){
    var totalStake = 0;
    async.waterfall([
        function(cb){
            MongoHandler.setAllBlockBalances(function(){
                cb();
            });
        },
        function(cb){
            if(suppliedBlockNumber){
                cb(null, suppliedBlockNumber);
            }
            else{
                MongoHandler.getCurrentBlockNumber(function(err, blockNumber){
                    cb(null, blockNumber);
                });
            }
        },
        function(endBlock, cb){
            var startBlock = endBlock - Constants.MAX_STAKEABLE_BLOCKS - Constants.MIN_HOLD_FOR_STAKE_BLOCKS;
            if(startBlock < 0){
                startBlock = 0;
            }
            var stakeStartBlock = endBlock - Constants.MAX_STAKEABLE_BLOCKS;
            if(stakeStartBlock < 0){
                stakeStartBlock = 0;
            }
            if(startBlock != stakeStartBlock - Constants.MIN_HOLD_FOR_STAKE_BLOCKS){
                stakeStartBlock = startBlock + Constants.MIN_HOLD_FOR_STAKE_BLOCKS;
                if(stakeStartBlock > endBlock){
                    return cb(); // 0 stake since minimum hold blocks not satisfied
                }
            }
            var blocks = [];
            var balances = {};

            for(var i = startBlock; i <= endBlock; i++){
                blocks.push(i);
            }

            BalanceCollection.find({
                $or : [
                    { "balances.user" : userId },
                    { "blockCreatorId" : userId }
                ],
                blockNumber : { $in : blocks }
            }, { $and : [
                { blockNumber : { $gte : startBlock } },
                { blockNumber : { $lte : endBlock } }
            ] },
            ).sort({blockNumber : 1}).toArray(function(err, docs){
                if(err){
                    console.log("Mongo Err: ", err);
                    return;
                }
                docs.forEach(function(data){ // Load users Balance in each block here
                    for(var i = 0; i < data.balances.length; i++){
                        if(data.balances[i].user == userId){
                            balances[data.blockNumber] = data.balances[i].balance;
                            break;
                        }
                    }
                });

                BalanceCollection.find({ "balances.user" : userId, blockNumber : { $lt : startBlock } }).sort({blockNumber : -1}).limit(1).toArray(function(err, lastBalance){
                    if(err){
                        console.log("Mongo Err: ", err);
                        return;
                    }
                    else if( (lastBalance || lastBalance.length || lastBalance[0].balances.length){
                        if(balances[startBlock] == null){ // If no transaction before startBlock, Balance = 0
                            balances[startBlock] = 0;
                        }
                    }
                    else{
                        if(balances[startBlock] == null){ // Load last balance as balance of startBlock
                            lastBalance[0].balances.forEach(function(data){
                                if(data.user == userId){
                                    balances[startBlock] = data.balance;
                                }
                            });
                        }
                    }
                });

                for(var i = startBlock + 1; i <= endBlock; i++){ // Load previous balance as current balance for all blocks without any transaction by given user
                    if(balances[i] == null){
                        balances[i] = balances[i-1];
                    }
                }

                docs.forEach(function(data){
                    if(data.blockCreatorId == userId){
                        var lastBlock = data.blockNumber + Constants.MIN_HOLD_FOR_STAKE_BLOCKS;
                        if(lastBlock > endBlock){
                            lastBlock = endBlock;
                        }
                        for(var i = data.blockNumber; i < lastBlock; i++){
                            balances[i] -= 1; // Remove 1 Stake-able coin from next hold blocks, since coin stake used for creation of current block
                        }
                    }
                });

                for(var i = stakeStartBlock; i <= endBlock; i++){ // Add all blocks stake
                    totalStake += balances[i];
                }

                cb();
            });
        },
        function(errs, results){
            callback(null, totalStake);
        }
    ], function(errs, results){
        //
    });
}

```

Figure 38. Coin-Age Calculation Function

5.3.10 Error Handling

The Miner and Wallet Nodes return error codes when a pre-defined error or exception occurs. These errors are conveyed to the End-User as a JSON response to the incoming HTTP request. The error responses always have the same format, only the error code number and its associated text message change between different error conditions. Figure 40 shows all the error codes and their respective error messages.

```
var ERROR_CODES = new Array();
/* ALL      */ ERROR_CODES[0] = '';
/* ALL      */ ERROR_CODES[1] = dataNotComplete;
/* ALL      */ ERROR_CODES[2] = dbError;
/* ALL      */ ERROR_CODES[3] = 'Redis Error';
/* ALL      */ ERROR_CODES[4] = 'User not Logged in !';

/* TRANSACTION */ ERROR_CODES[5] = 'Transaction Data Missing';
/* TRANSACTION */ ERROR_CODES[6] = 'Invalid Transaction Amount';
/* TRANSACTION */ ERROR_CODES[7] = 'Invalid Transaction Fees';
/* TRANSACTION */ ERROR_CODES[8] = 'Invalid Sender Address';
/* TRANSACTION */ ERROR_CODES[9] = 'Invalid Receiver Address';

/* TRANSACTION */ ERROR_CODES[10] = 'Invalid Private Key';
/* TRANSACTION */ ERROR_CODES[11] = 'Given Deadline is invalid';
/* TRANSACTION */ ERROR_CODES[12] = 'Error Generating Nonce';
/* TRANSACTION */ ERROR_CODES[13] = 'Invalid Nonce / Nonce too small';
/* TRANSACTION */ ERROR_CODES[14] = 'Invalid txId (Transaction Hash does not match)';
/* TRANSACTION */ ERROR_CODES[15] = 'Signature Mismatch / Unauthenticated Transaction';
/* TRANSACTION */ ERROR_CODES[16] = 'Sender and Receiver cannot be the same';

/* BLOCK      */ ERROR_CODES[20] = 'Invalid userId';
/* USER       */ ERROR_CODES[30] = 'Invalid Key pair';
```

Figure 39. Error Codes for Invalid API Calls

This enables End-Users and other developers to recognize the precise error that occurred. The End-User can avoid this error in the next API calls by modifying appropriate values in the input object.

5.3.11 Account Balance Calculations

The account balance needs to be calculated in multiple steps throughout the Mining process. It is also required during the Coin-Age calculations and when the user requests for their current balance.

In Figure 41 below we match all the coins sent by the Wallet holder and all the coins received along with all the coins earned via Mining. We then calculate his balance as the sum of all the coins received and all the coins earned as block rewards minus the number of coins spent in past transactions.

```

calculateAccountBalanceTillBlock : function(userId, blockNumber, callback){
  var matchSenderQuery = {
    "transactions.sender" : userId
  };

  var matchReceiverQuery = {
    "transactions.receiver" : userId
  };

  var matchCreatorQuery = {
    "blockCreatorId" : userId
  };

  if(blockNumber || blockNumber === 0){
    matchSenderQuery.blockNumber = { $lte : blockNumber };
    matchReceiverQuery.blockNumber = { $lte : blockNumber };
    matchCreatorQuery.blockNumber = { $lte : blockNumber };
  }

  async.parallel([
    function sentCoins(cb){
      BlockCollection.aggregate([
        {
          $match : matchSenderQuery // This match will reduce the number of unwind operations
        },
        {
          $unwind : "$transactions"
        },
        {
          $match : {
            "transactions.sender" : userId
          }
        },
        {
          $group : {
            _id : "$transactions.sender",
            amount : { $sum : { $multiply: [ "$transactions.amount", Constants.SUM_DECIMAL_CORRECTION ] } },
            fees : { $sum : { $multiply: [ "$transactions.fees", Constants.SUM_DECIMAL_CORRECTION ] } }
          }
        }
      ], function(err, docs){
        cb(err, docs[0] ? (docs[0].amount + docs[0].fees) / Constants.SUM_DECIMAL_CORRECTION : 0);
      });
    },
    function receivedCoins(cb){
      BlockCollection.aggregate([
        {
          $match : matchReceiverQuery // This match will reduce the number of unwind operations
        },
        {
          $unwind : "$transactions"
        },
        {
          $match : {
            "transactions.receiver" : userId
          }
        },
        {
          $group : {
            _id : "$transactions.receiver",
            amount : { $sum : { $multiply: [ "$transactions.amount", Constants.SUM_DECIMAL_CORRECTION ] } }
          }
        }
      ], function(err, docs){
        cb(err, docs[0] ? (docs[0].amount) / Constants.SUM_DECIMAL_CORRECTION : 0);
      });
    },
    function earnedFees(cb){
      BlockCollection.aggregate([
        {
          $match : matchCreatorQuery // This match will reduce the number of unwind operations
        },
        {
          $group : {
            _id : "$blockCreatorId",
            amount : { $sum : { $multiply: [ "$totalFees", Constants.SUM_DECIMAL_CORRECTION ] } }
          }
        }
      ], function(err, docs){
        cb(err, docs[0] ? (docs[0].amount) / Constants.SUM_DECIMAL_CORRECTION : 0);
      });
    },
    function(errs, amounts){
      callback(null, amounts[1] + amounts[2] - amounts[0]); // received + fees earned - sent
    });
  ],
),

```

Figure 40. Account Balance Calculation Function

5.3.12 Target Difficulty Calculations

The target value for the network difficulty needs to be calculated for every `DIFFICULTY_CHANGE_EVERY_BLOCKS` number of blocks, as defined in the `constants.js` file. The target difficulty is used to verify all the incoming blocks and self-generated blocks contain a valid proofHash and is responsible for bringing the network into a uniform state without requiring a centralized authority.


```

setAllBlockTargets : function(callback){
  MongoHandler.getCurrentBlockNumber(function(err, currentBlockNumber){
    TargetCollection.find({}).toArray(function(err, docs){
      var targetExist = [];
      var addTarget = [];
      if(err){
        console.log("MongoErr: ", err);
      }
      docs.forEach(function(doc){
        targetExist.push(doc.blockNumber);
      });
      for(var i = 0; i <= currentBlockNumber-1; i = i + Constants.DIFFICULTY_CHANGE_EVERY_BLOCKS){ // +1 to precalculate incase next block needs new target
        if(targetExist.indexOf(i) == -1){
          addTarget.push(i);
        }
      }
      async.eachSeries(addTarget, function(blockNumber, cb){
        var targetObj = {
          blockNumber : blockNumber
        };
        if(blockNumber == 0){
          targetObj.target = Constants.GENESIS_BLOCK_TARGET;
          TargetCollection.insert(targetObj, function(err, result){
            cb();
          });
        }
        else{
          blockCollection.find({ $or : [
            { blockNumber : blockNumber - 1 }, // -1 since next block may not exist yet
            { blockNumber : blockNumber - Constants.DIFFICULTY_CHANGE_EVERY_BLOCKS }
          ]}).toArray(function(err, docs){
            if(docs[0].timestamp > docs[1].timestamp){
              console.log("Missing timestamp while calculating Target"); // TODO : Handle properly
              cb();
            }
            var timeDifference = docs[1].timestamp - docs[0].timestamp;
            TargetCollection.find({blockNumber : blockNumber - Constants.DIFFICULTY_CHANGE_EVERY_BLOCKS}).toArray(function(err, targetDocs){
              if(err){
                return console.log("MongoDB error: ", err);
              }
              var prevTarget = targetDocs[0].target;
              var maxDifficulty = parseInt("ffffff", 16);

              var oldZeros = parseInt(prevTarget.substring(0,2), 16);
              var oldDifficulty = parseInt(prevTarget.substring(2), 16);
              var newZeros = oldZeros;

              var newDifficulty = oldDifficulty * timeDifference / (Constants.AVERAGE_BLOCK_TIME_MS * Constants.DIFFICULTY_CHANGE_EVERY_BLOCKS);

              console.log("newDifficulty", newDifficulty);
              while(newDifficulty > maxDifficulty){
                newZeros++;
                newDifficulty = newDifficulty / 256;
              }
              var newTarget = parseInt(newDifficulty).toString(16);
              while(newTarget.length < 6){
                newTarget = "0" + newTarget;
              }
              var newZerosStr = newZeros.toString(16);
              while(newZerosStr.length < 2){
                newZerosStr = "0" + newZerosStr;
              }
              console.log("New target : ", newZerosStr + newTarget);
              targetObj.target = newZerosStr + newTarget;
              TargetCollection.insert(targetObj, function(err, result){
                cb();
              });
            });
          });
        }
      }, function(errs, results){
        callback();
      });
    });
  });
}

```

Figure 41. Set Target Values for all Previous Blocks Function

The setAllBlockTargets function in Figure 42 calculates target values for all the previous blocks and caches it into a MongoDB collection. This value generally does not change once it has been calculated, thus it can be stored and reused for the next target value calculation every “DIFFICULTY_CHANGE_EVERY_BLOCKS” number of blocks.

5.3.13 Caching Account Balances for Performance

The Miner needs to check a user's balance when calculating the Coin-Age, and this is a slow process due to the complex query performed during the Coin-Age calculation. We can speed up this process by caching the balances for each previous block. This value can be cached since the user's balance in any past block will always remain constant. Thus, we cache user's balances for each block and use these for the Coin-Age calculations, helping us significantly increase the Miner Node's performance.

```

setBlockBalances : function (blockNumber, callback) {
  BalanceCollection.find({blockNumber : blockNumber}).toArray(function(err, docs){
    if(docs && docs.length == 0){
      BlockCollection.find({blockNumber : blockNumber}).toArray(function(err, doc){
        if(err || !doc || !doc.length){
          console.log("No Such Block : ", blockNumber);
          callback();
        }
        else{
          var block = doc[0];
          var transactions = block.transactions ? block.transactions : [];
          var users = [block.blockCreatorId];
          var balances = [];

          transactions.forEach(function(transaction){
            if(users.indexOf(transaction.sender) == -1){
              users.push(transaction.sender);
            }
            if(users.indexOf(transaction.receiver) == -1){
              users.push(transaction.receiver);
            }
          });

          async.each(users, function(userId, cb){
            MongoHandler.calculateAccountBalanceTillBlock(userId, blockNumber, function(err, balance){
              balances.push({
                user : userId,
                balance : balance
              });
              cb();
            });
          });

          var balanceObj = {
            blockNumber : blockNumber,
            blockHash : block.blockHash,
            blockCreatorId : block.blockCreatorId,
            balances : balances
          };
          BalanceCollection.insert(balanceObj, function(err, result){
            if(err){
              console.log("Mongo Insert Error");
            }
            callback();
          });
        }
      });
    }
    else{
      console.log("Duplicate Balance Insert : Block Number : " + block.blockNumber + " BlockHash : " + block.blockHash);
    }
  });
},

```

Figure 42. Block Balance Caching Function for Coin-Age Calculation

It is important to not use this cache during the balance calculation step of the transaction validation as this value may be stale due to forks in the blockchain. But it can be used safely for the Coin-Age (stake) calculations.

6. CONCLUSION AND FUTURE SCOPE

To conclude, eCash.js is a complete cryptocurrency built from scratch using JavaScript, thus can be mined on any device with minimal overhead. It uses a combination of Proof-of-Work and Proof-of-Stake for consensus to provide better protection to 51% attacks.

This project requires a lot of testing over a large network of nodes to identify and fix any bugs or vulnerabilities before its release deployment. Due to limited computational resources I was only able to perform testing over a network of 3 computers. The future scope of this application may include some of the following:

1. Pool mining with profit distribution (Mining coins on the Browser)
2. Peer-to-Peer discovery, making it independent of static IPs
3. Testing on a larger number of Full Nodes to find better constants.js values
4. Introduction of smart-contracts

As described in the initial project requirements, eCash.js contains:

1. Ability to create a new Public/Private Key pairs (Wallet address)
2. A blockchain which maintains all the transactions and acts as a Distributed Ledger
3. Ability to update the blockchain and manage forks in the blockchain
4. Send and receive coins.
5. Ability to check any wallet's balance, using its public key.
6. Ability to connect to other Full Nodes via the internet.
7. Miner Software, which will process all the unconfirmed transactions

Bibliography

- [1] V. Beal, "What is Entity Relationship Diagram?," Webopedia Definition. [Online]. Available:
https://www.webopedia.com/TERM/E/entity_relationship_diagram.html. [Accessed 26 March, 2019].
- [2] Motea Alwan, " What is System Development Life Cycle?," Airbrake Blog. [Online]. Available: <https://airbrake.io/blog/sdlc/what-is-system-development-life-cycle>. [Accessed 26 March, 2019].
- [3] The Editors of Encyclopaedia Britannica, "Client-server architecture," Encyclopaedia Britannica. [Online]. Available:
<https://www.britannica.com/technology/client-server-architecture>. [Accessed 26 March, 2019].
- [4] The Editors of Techopedia, "three-tier-architecture," Techopedia. [Online]. Available: <https://www.techopedia.com/definition/24649/three-tier-architecture>. [Accessed 26 March, 2019].
- [5] The Editors of Wikipedia, "Software design pattern," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Software_design_pattern. [Accessed 26 March, 2019].

- [6] The Editors of Tutorialspoint, "Design Patterns - MVC Pattern," Tutorialspoint. [Online]. Available: https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm. [Accessed 26 March, 2019].
- [7] The developers of Amazon, "Elastic Compute Cloud (EC2) - Cloud Server & Hosting - AWS," Amazon, [Online]. Available: <https://aws.amazon.com/ec2/>. [Accessed 26 March, 2019].
- [8] Jake Frankenfield, "Proof of Stake (PoS)," Investopedia, 07/30/2019. [Online]. Available: <https://www.investopedia.com/terms/p/proof-stake-pos.asp>. [Accessed 26 March, 2019].
- [9] The Editors of Wikipedia, "Proof-of-stake," Wikipedia. [Online]. Available: <https://en.wikipedia.org/wiki/Proof-of-stake>. [Accessed 26 March, 2019].
- [10] The developers of Node.js, "Node.js Homepage," Node.js. [Online]. Available: <https://nodejs.org/>. [Accessed 26 March, 2019].
- [11] The developers of MongoDB, "What is MongoDB ?," MongoDB. [Online]. Available: <https://www.mongodb.com/what-is-mongodb>. [Accessed 26 March, 2019].
- [12] The developers of Redis, "Redis Homepage," Redis. [Online]. Available: <https://redis.io/>. [Accessed 26 March, 2019].