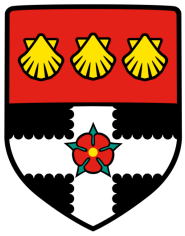


University of Reading

Department of Computer Science

A Java Based Blockchain Network Implementing the CryptoNight Proof of Work Algorithm

Jounaid Ruhomaun

Supervisor: Dr Hong Wei

A report submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Bachelor of Science in *Computer Science*

April 29, 2021

Declaration

I, Jounaid Ruhomaun, of the Department of Computer Science, University of Reading, confirm that all the sentences, figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

Jounaid Ruhomaun
April 29, 2021

Abstract

The blockchain technology has emerged as one of the most discussed topics throughout the 21st century, not only within the computing discipline, but also with the general public. Its implementation in the P2P electronic cash systems, known as cryptocurrencies, changed the way many people viewed the transfer of wealth. However, as the technology grew in popularity, and with the advent of ASIC and GPGPU hashing, issues surrounding general users abilities to compete in the process became prevelant, which threatened the integrity and core principles that these networks were built upon. This paper documents the implementation of a P2P blockchain network implementing the CryptoNight proof of work algorithm, which is designed to mitigate these emergent problems that current blockchains face. Throughout development, priority has been placed on utilising modern frameworks, standards and techniques, and as such the Java Spring framework with OpenAPI integration was chosen, and a RESTful API is provided. Two Java implementations of the CryptoNight proof of work algorithm were developed and published, with performance testing carried out in comparison to other language solutions. Manual integration and component testing of the P2P blockchain server application utilising an AWS EC2 cloud instance was carried out, with the results documented and discussed. An analysis of the achievements and solutions produced within the project is given, with a list of future works that are to be carried out also being provided.

Keywords: Blockchain, CryptoNight, P2P Network, RESTful API

Report's total word count: 19,783

Contents

List of Figures

List of Tables

List of Abbreviations

P2P	Peer to Peer
ASIC	Application Specific Integrated Circuit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
REST	Representational State Transfer
API	Application Programming Interface
AWS	Amazon Web Services
EC2	Elastic Compute Cloud
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
HTML	Hyper Text Markup Language
TSS	Time Stamping Service
ECC	Elliptic Curve Cryptography
RSA	Rivest Shamir Adleman (Cryptography)
BTC	Bitcoin
SHA	Secure Hashing Algorithm
LTC	Litecoin
GUI	Graphical User Interface
XMR	Monero
NFT	Non Fungible Token
ETH	Ether
GDDR5	Graphics Double Data Rate 5
L3	Level 3 (cache)
MB	Megabyte
CRUD	Create, Read, Update, Delete
JVM	Java Virtual Machine
JNI	Java Native Interface
NIST	National Institute of Standards and Technology

AES	Advanced Encryption Standard
FIPS	Federal Information Processing Standards
OS	Operating System
POM	Project Object Model
GPG	GNU Privacy Guard
YAML	YAML Ain't Markup Language
JSON	JavaScript Object Notation
IANA	Internet Assigned Numbers Authority
IPV4	Internet Protocol Version 4
IPV6	Internet Protocol Version 6
SLF4J	Simple Logging Facade for Java
UML	Unified Modeling Language
BDD	Behaviour Driven Development
CI	Continuous Integration

Chapter 1

Introduction

1.1 What is Blockchain?

The blockchain technology essentially consists of a distributed ledger, with the purpose of instilling trust and consensus around a particular digital asset within a trustless decentralised environment. Said ledger consists of data constructs referred to as 'blocks' which each contain: a timestamp; a field containing the digital asset in question; a cryptographic hash of all data within the block; a reference to the hash of the previous block in the ledger; some form of consensus protocol/algorithm, and various other meta properties. A visualisation of what this ledger looks like can be seen in Figure 1.1.

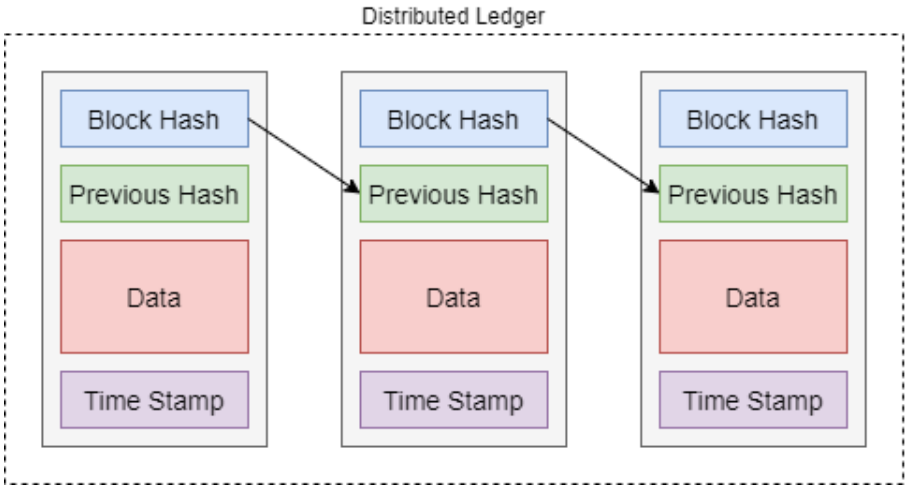


Figure 1.1: Blockchain ledger overview

A consensus system is required to produce the next block in the ledger, which generally involves some form of cost or difficulty in order to satisfy a certain condition. However, said system must also be easily verifiable by members of the distributed network in order for consensus of the new generated block to be quickly gained. A reward is given to the user who generates the block, which acts as an incentive for users to contribute towards the consensus system.

This system of consensus, in combination with the fact that every block in the ledger is linked through their hashes makes it incredibly difficult, if not impossible to tamper with any one block in the ledger. This is because the consensus algorithm/protocol must be recalculated for every other block in the ledger in order reproduce valid linking hashes, with the next block in the ledger usually being produced before this could happen.

1.2 Problems With Current Blockchains

This project targets two problems that face current blockchain implementations. The first of which involves the most common type consensus system implemented within blockchains, the proof of work algorithm. This algorithm generally consists of a cryptographic puzzle, where a hash of the block with a defined number of proceeding zeros must be found. This can only be done through calculating said hash consecutively until a satisfactory result is found, which requires large amounts of compute. The complexity of the puzzle can be increased simply by required more proceeding zeros for a satisfactory hash, which is referred to as the proof of work difficulty. This allows the complexity of the puzzle to scale with the increase in total hashing power that results from a growing user base within the network.

The principle behind this consensus algorithm is that said algorithm should only require arbitrary mathematical complexity, and there should be an equal chance for all individuals attempting the puzzle to produce a solution. As is quoted in the Bitcoin whitepaper, the first blockchain project to implement the such an algorithm, 'proof of work is essentially one CPU, one vote' (Nakamoto, 2008). However, as blockchain implementations began to gain popularity, hardware that is specifically designed to efficiently solve said algorithms (referred to as ASICs) were developed. This put most users with general CPUs at a huge disadvantage when competing to solve the proof of work puzzle, which discourages many users from contributing to the network. This is a problem as in order for the network to function, there needs to be a level of distribution between users, as if one individual or organisation controls more than 51% of the networks hashing power, they can start manipulating blocks (known as the 51% attack, Sayeed and Marco-Gisbert (2019)) which destroys the integrity of the network.

One solution to this problem is the CryptoNight proof of work algorithm, which was originally defined within the CryptoNote whitepaper (Van Saberhagen, 2013). The algorithm is designed to be dependant on read/writes to memory, which introduces inefficiency on ASICs and GPGPUs, whilst having proportionally less of an impact on CPUs due to the cache.

The second problem explored within this project is to do with how the server applications within most blockchain implementations are actually developed. Most are designed with a large degree of complexity and application specific protocols. This makes it difficult for developers to not only contribute to the open source projects, but also create third party components such as wallet applications. A solution to this problem would be to design a server application implementing a RESTful API which works over HTTP, as this is currently the standard communication framework used in the majority of modern web applications.

1.3 Project Objectives

Given the two solutions identified for the problems discussed, the objectives for this project are as follows:

1. Develop a Java implementation of the CryptoNight algorithm as there are currently **no published Java solutions** (prior to this project).
2. Integrate the CryptoNight component developed into a P2P blockchain server application, that provides a RESTful API and is developed with modern frameworks, technologies and best practice.

These objective are expanded upon within the requirements analysis section of this report (see section 3.1).

1.4 Summary of Repositories and Materials Produced

The following repositories and artifacts have been produced in relation to the CryptoNight component of this project:

- **https://github.com/jounaidr/CryptoNightJ** - *repository containing the complete code for the CryptoNightJ solution.*
- **https://github.com/jounaidr/keccak-java-speedtest** - *a repository for performance testing different Java Keccak implementations, with the purpose of optimising the CryptoNightJ solution.*
- **https://github.com/jounaidr/CryptoNightJNI** - *a repository containing the CryptoNightJNI wrapper solution bindings with the C++ binaries used.*
- **https://mvnrepository.com/artifact/com.jounaidr/CryptoNightJNI** - *released CryptoNightJNI artifacts available on the Maven central repository.*

The following repositories and documentations have been produced in relation to the P2P blockchain server application component of this project:

- **https://github.com/jounaidr/jrc-node** - *a repository containing the complete code for the blockchain server application.*
- **https://jounaidr.github.io/jrc-node-javadocs/** - *a website hosting HTML formatted JavaDoc documentation of the P2P blockchain server application.*
- **https://jounaidr.github.io/jrc-node-API-docs/** - *a website hosting HTML formatted Swagger generated API documentation for the P2P blockchain server application.*

1.5 Organization of the Report

This report is organised into six chapters. Chapter 2 involves a review of the research and literature that contributed towards the development of the blockchain technology, as well as an analysis of the notable blockchain implementations, and the main technologies used within this project.

Chapter 3 provides a detailed look into the implementation of the various components that make up this project. Section 3.1 defines the project requirements and provides a short analysis of the solutions proposed. Section 3.2 details the design and implemenation of the CryptoNight algorithm, and the Java solutions produced. Section 3.3 provides an in depth record of the implementation of the P2P blockchain server application.

Chapter 4 gives an overview and discussion of the results achieved within the project. Section 4.1 provides performance statistics for the CryptoNight solutions developed, in comparison to other language implementations. Section 4.2 details the manual integration and component testing that was carried out on the blockchain server application.

Chapter 5 summarizes the project with a conclusion in reference to the initial requirements, as well as providing a list of future works that will be implemented. Finally, chapter 6 provides a short reflection on the learning experiences that transpired during the development of this project, as well as a justification for the deviations from the project's initial documentation that occurred.

Chapter 2

Literature and Technology Review

The following section details an overview of the literature contributing to the inception of blockchain, the current prominent technologies and implementations that utilise blockchain, as well an analysis of the research that lead to the development of the CryptoNight proof of work algorithm. A review of the original literature behind the RESTful framework is also provided.

2.1 The Inception Of Blockchain

The history of blockchain dates back to the early 1990's, where the idea of having trust within a distributed network with no central authority was explored by Haber and Stornetta (1991). Later developments in the field were made by Nick Szabo, with the idea of 'bit gold', which laid out many of systems and protocols that make up modern blockchains and cryptocurrencies. The idea of chaining together multiple exclusive timestamped events through the use of cryptography was explored by Stefan Konst in the early 2000's. All of these ideas and theories culminated in the invention and implementation of the first commercial blockchain, Bitcoin (Nakamoto, 2008).

2.1.1 How To Time-Stamp a Digital Document (1991)

The ideas that went on to form the design and invention of the modern blockchain framework were first conceptualised within the Journal Of Cryptology paper titled: 'How To Time-Stamp a Digital Document' by Haber and Stornetta (1991). The paper presents an issue around the time stamping of digital media (in particular documents), being the fact that said timestamps can be easily tampered with. An example is given with the importance of time stamping intellectual property, where it is imperative an inventor can verify when they first documented an idea in order to mitigate competing claims of originality.

The paper suggests two properties a time stamp would have to posses in order to be trustworthy. First, the data of the digital media itself must be timestamped, not the medium it's contained within. Second, its must be impossible to stamp said digital media with a time and date different from its actual timestamp. This alludes to the use of cryptography, as vast amounts of data can converted into an irreversible and unique digital signature.

Two potential solutions to this problem are provided, both through the use of a one-way cryptographic hash. The scenario presented is having multiple users within a distributed network, where each user requires the ability to timestamp their own documents, verify the timestamps and authenticity of other users documents, whilst also having assurance of the validity of said timestamps date and time (with the timestamps being unchangeable hence the cryptographic hash).

A system where each user simply hashes their own documents would not meet the scenarios criteria, as despite the hashed timestamp being unchangeable, users could have provided incorrect time and date information prior to hashing. Therefore a solution is suggested in where documents are to be submitted to a time-stamping service (TSS) authority which produces the hashes, and links said hashes together to provide a 'certificate' of those documents timestamp authenticity, which is subsequently distributed to the network.

There are many issues with this centralised model however, the first of which being the fact that the privacy/security of documents submitted to the TSS is compromised, as users are no longer in exclusive possession of their document. This leads into the second issue of incompetence and trust, as users are relying on a central authority which could potentially collude with certain clients to forge timestamps, or simply just loose or incorrectly store documents within the their possession, invalidating said documents certification. The final

issue presented suggests the amount of bandwidth and storage that would be required by the TSS could incur high costs prohibiting such models from operating (this issue was more prevalent in 1991).

The second solution presents a way in which a TSS is not required. This is done by having an automated system that selects users within the network to hash documents based on a pseudo random number generator (using the documents themselves as the seed), which results in a distributed sense of trust and consensus.

This notion and method of trust within a distributed network absent of a central authority is what was later developed into some of the standards that form the modern blockchain framework.

2.1.2 Bit Gold (1998)

Computer scientist Nick Szabo was recorded to have made significant contributions to the ideas behind blockchain with the concept of 'Bit Gold' in 1998 (Szabo, 2008). Szabo conceptualises that a practical use for a system of decentralised trust within a trust-less environment would be currency, as relying on third party authorities to control such an important part of capitalistic societies is not ideal, with multiple examples of hyperinflation and monetary irresponsibly by said authorities being recorded in recent times.

Szabo comments on a previous medium of exchange which had no reliance on a third party, being that of precious metals (such as gold), where the objects of exchange themselves have value due to their rarity. However, obvious issues around such mediums such as impractically, and more recently, the inability to use such mediums through online transactions result in said mediums being unusable as currency in modern times. Therefore Szabo conceptualised the idea of having 'unforgeable costly bits' as a medium of exchange, likened to a form of digital gold, hence the name bit gold.

The concept of having a computationally difficult 'puzzle' in order to generate bit gold as suggested by Szabo is what would go on to coin the term 'block mining' in modern blockchains. Having the stings of bit gold (as well as proof of generation with said hashing puzzle) submitted to a 'distributed property title registry' allows for consensus between bit gold users, since they can verify that strings of bit gold belong to their respective owners by simply checking said registry.

A potential problem with bit gold that Szabo comments on is the fact that the described 'puzzles' that are responsible for producing bit gold are reliant on computer architecture, not just abstract mathematical complexity. Therefore it would be possible for some users to become low cost producers of bit gold, potentially flooding the market.

These remarkably early predictions made by Szabo became real issues during later development and deployment of actual blockchains, which lead to the concept of adjustable difficulty within proof of work puzzles.

2.1.3 Theories On Cryptographically Secured Chains (2000)

Stefan Konst (2000) made significant contributions to the field with his theories on cryptographically secured chains in his research paper titled: 'Secure log files based on cryptographically linked entries' (translated). Konst comments on the need to not only verifying the authenticity of log events, but also the order in which said log events happen in. Therefore he suggests a way of 'chaining' said entries cryptographically so that any one link in the chain cannot be compromised.

2.2 Cryptocurrencies

The blockchain technology is somewhat synonymous with cryptocurrencies, and rightfully so as an overwhelming majority of mature blockchain projects currently in active use are cryptocurrencies, as well as the first blockchain implementation also being conceived for said application. The term 'cryptocurrency' has been coined as an abbreviation for cryptographic currencies, which alludes to how said currencies are created, secured and transferred within the trustless blockchain environment, with strong cryptography.

The basis of cryptocurrencies start with the idea of 'wallets', which essentially consists of a key pair generated through public key cryptography. These key pairs allow for individuals to cryptographically 'sign' strings of text, for which said signatures can be verified with the public key ensuring the signatures authenticity (since the private key is only known to the wallet holder). The cryptographic technique that is generally used for said key pairs within cryptocurrencies is elliptic curve cryptography (ECC), based on elliptic curve theory. The benefits of said public key cryptography against the more widely used RSA (Rivest–Shamir–Adleman) is that the same level of security is provided whilst also having a significantly lower key size (256 bits as apposed to the 3072 bits of RSA keys, Wagner (2021) suggests). This simplifies the sharing and recording of public keys (known as wallet addresses) allowing for easier transactions between individuals.

Users of the network can generate transactions which denote an amount of funds to be sent, and the wallet address said funds are to be sent to (along with an optional user defined network fee). The transaction is signed before being submit to a nodes 'transaction pool'. Said transaction pool is shared amongst all nodes within the network, for which miners of the blockchain can select transactions to include within the block they are attempting to mine (generally a 'greedy' approach is taken, where transactions with the highest network fees are selected). Once a successful block has been mined, transactions within said block are confirmed, with each subsequent block mined adding an additional confirmation, which gives users a certain level of assurance that a transaction is legitimate. Users can query a nodes blockchain ledger to trace all the transactions relating to a particular wallet address, with the sum of all transactions (sent and received) resulting in the wallets balance.

Currency is injected into the network whenever a block is mined, with the miner earning a defined amount of currency along with the summation of all network fees within the blocks' transactions.

2.2.1 Bitcoin (BTC)

Bitcoin is truly where this world of blockchain really began, being the first blockchain and cryptocurrency implementation. Its features were outlined in Nakamoto's famous whitepaper titled: 'Bitcoin: A Peer-to-Peer Electronic Cash System', which formalised many of the previous ideas in the field, such as Haber and Stornetta's digital time stamping and Szabo's Bit Gold, as well as the concept of wallets and transactions as discussed previously. The hashing algorithm used for the proof of work is SHA-2, as this was the standard cryptography function generally in use around the time of Bitcoin's inception. Seven principles (*Principles of Bitcoin*, 2017) were defined for Bitcoin which contributed to its reliability and overall success:

- **Maximum number of coins** - *there is a finite number of coins that will be genenerated in the network, set to 21 million.*
- **Zero censorship** - *absolutely no one can prevent a transaction from being confirmed.*

- **Open source** - *the source code for Bitcoin must all ways remain public with complete transparency.*
- **Permissionless** - *absolutely everyone can partake in all aspects of the network with no specified criteria required.*
- **Pseudonymity** - *there is no personally identifiable information required to partake in the network.*
- **Fungibility** - *each Bitcoin generated in the network is of equal value.*
- **Immutability** - *the blockchain ledger and transaction history is unchangeable.*

After Bitcoin's popularization, these basic principles were used to create many different alternate versions of the Bitcoin project, which are referred to 'altcoins'. Each altcoin brought some from of improvment or innovation to the field.

2.2.2 Litecoin (LTC)

Litecoin was one of the original altcoins which forked the Bitcoin source code (Lee, 2011). Litecoin was designed for use in smaller transactions, reducing the time a block was generated in to 2.5 minutes, from the 10 minutes used in Bitcoin. It also replaced the proof of work algorithm, adopting scrypt in place of the SHA-2 algorithm used in bitcoin. Other than this, minor changes were made to the maximum supply of coins and small modifications to the GUI.

2.2.3 Monero (XMR)

One of the most notable innovations in the field was with the inception of Monero, which was conceptualised within Van Saberhagen's 2013 whitepaper titled: 'CryptoNote V2.0'. Monero focuses on two main principles that are deficient in Bitcoin, privacy and fairness.

All though Bitcoin wallets and transactions require no personal identification to create, the wallet addresses themselves, as well as said wallets transaction history is publicly available and traceable through the public ledger. During the early days of Bitcoin this wasn't much of an issue since transactions generally happened between miners, however as Bitcoin and other cryptocurrencies popularity increased with the general public, large cryptocurrency exchange platforms such as Coinbase were established. These exchange platforms eventually became heavily regulated by government, and generally require multiple forms of photo identification in order to set up an account which links an individuals identity to their wallet and transaction history. Monero ensures privacy by utilising stealth addresses with ring signatures, as well as having an obfuscated public ledger to hide transaction history.

One of the main innovations Monero introduced into the field was with the CryptoNight proof of work algorithm. This algorithm, unlike SHA-2 was designed with sufficient complexity so that the algorithm could be more efficiently solved on general CPUs in comparison to other more specific hardware. This made it much easier to compete in the proof of work system as a general user, as this cryptographic puzzle was more true to the 'one CPU one vote' principle outlined in Bitcoin.

2.3 Non Fungible Tokens

The latest development in the blockchain technology is the advent of Non Fungible Token (NFT) blockchains (Kramer, 2021). Unlike cryptocurrencies where each asset transferred within the blockchain is fungible and of the same value, NFTs can have vastly differing value dependant on the specific asset it represents. Any unique asset of value can be represented and traded as an NFT, examples include property and artwork. NFTs open up a world of possibilities in terms of digital media, where despite media such as graphic artwork or music files being replicable, an NFT assigned to said media is not and therefore can be allocated value and traded. An example of a popular NFT blockchain is CryptoKitties (Dapper Labs, n.d.), where unique 'kitten' assets are generated as NFTs which can be traded amongst users, with highest value NTF within the blockchain being sold for 600ETH (equivalent to aprox 1.2 million GBP as of April 2021).

2.4 The CryptoNight Proof of Work Algorithm

As mentioned previously, the CryptoNight algorithm was first conceptualised within the CryptoNote whitepaper (Van Saberhagen, 2013) and was first implemented within the Monero cryptocurrency. The paper suggests that a true proof of work algorithm should have a liner relationship between the chance a successful proof of work is found, and electrical power costs. However, due to ASICs and GPGPUs this is not the case, with such hardware being able to calculate the proof of work much more efficiently than a general CPU with the same power investment. It is mentioned in the paper that because of this, generally 80% of a blockchain's networks hashing power is controlled by 20% of the networks user base. Not only does this go against the principles of distribution, but also discourages new users from attempting to contribute in the network.

Therefore the CryptoNight algorithm was proposed, which as mentioned introduces inefficiency for such hardware by establishing a dependency on system memory. A 2MB allocation within the algorithm is suggested for the following reasons:

- Such a size fits nicely within the L3 cache of most modern CPUs (at the time of this algorithms inception)
- Internal memory of greater than 1MB is an unacceptable size for modern ASIC pipelines
- All though GPU GDDR5 memory has remarkable bandwidth, it is much slower than CPU L3 cache

Therefore, an algorithm designed with such a memory dependency mitigates the issues seen in Bitcoin surrounding proof of work fairness when ASICs and GPUs are introduced into the network.

2.5 RESTful APIs

The Representational State Transfer (REST) framework was originally defined within Fielding's famous dissertation titled: 'Architectural Styles and the Design of Network-based Software Architectures'. Fielding proposed a method for interaction between distributed system over HTTP that is flexible and resource independent, unlike the previous SOAP standard that was tied to XML format. As is suggested, this allows for separate and unique software to communicate through a common set of rules defined in the REST API. There are four HTTP

request methods defined within the REST architecture, that are based on the CRUD concept (create, read, update and delete, Martin 1983):

- **GET** - *request the retrieval of data specified by the endpoint*
- **POST** - *request the creation of new data at the specified endpoint*
- **PUT** - *request data at the specified endpoint to be updated*
- **DELETE** - *request the deletion of data at the specified endpoint*

In combination with these request methods, Fielding and Reschke suggested a defined set of standard 'status codes' should be used in response to said requests. The status codes proposed are as follows:

- **100 range** - *relays basic information, such as 'continue'*
- **200 range** - *denotes a successful request*
- **300 range** - *notifies a redirect, such as 'moved permanently' or temporary redirect'*
- **400 range** - *denotes a client side error, such as 'bad request' or 'unauthorized'*
- **500 range** - *denotes a server side error, such as 'internal server error' or 'bad gateway'*

These proposed rules eventually became the generally accepted standard for communication between web applications, with almost all modern web applications nowadays implementing a RESTful approach.

2.6 Summary

In summary, this chapter first provides an overview of the literature and research that contributed to the inception the blockchain technology, as well as an analysis of the current prominent blockchain implementations. A detailed look into the literature behind the two specific technologies explored within this project, the CryptoNight proof of work algorithm, and the RESTful architecture, is also provided.

Chapter 3

Methodology

3.1 Requirement Analysis

The requirements for this project are written in reference to the paper titled: 'Writing Good Requirements' published within the Proceedings of the Third International Symposium of the INCOSE journal (Hooks, 1994). The paper suggests that requirements should be written to be necessary, verifiable and attainable. A necessity in the requirement means that implementation of said requirement is imperative towards the success of the project. A Verifiable requirement means its written with a certain acceptance criteria. The requirement must also be attainable within the restrictions of the project. With this in mind, the following requirements have been formulated:

1. *A P2P blockchain server application shall be developed.*
    1.1. *The P2P blockchain server application shall implement a proof of work algorithm that doesn't require specific hardware to be solved efficiently.*
    1.2. *The P2P blockchain server application shall provide non implementation/language specific protocols and APIs.*
    1.3. *The P2P blockchain server application shall reject invalid blocks.*
    1.4. *The P2P blockchain server application shall have dynamic difficulty adjustment based on the hashing power of the network.*
    1.5. *The P2P blockchain server application shall not be dependant on system architecture.*
    1.6. *The P2P blockchain server application shall be able to handle multiple peers simultaneously.*
    1.7. *The P2P blockchain server application shall have automatic peer discovery.*
2. *A Java solution for the CryptoNight hashing algorithm that is system independent shall be developed*
    2.1. *The Java CryptoNight solution should be optimal in comparison to to other CryptoNight implementations.*
    2.2. *The Java CryptoNight solution should be modular and usable within other Java applications.*

Requirements 1.1 and 1.2 denote the main innovation to the project and therefore are paramount in the the projects success.

In order to satisfy requirement 1.2, a RESTful API is chosen for which the implementation of will be done through OpenAPI generator (OpenAPI, 2015) with Spring integration. The reason such tool is chosen is that the implementation of the API is done in non language specific specification file, that can then be used to generate client/server code in almost any language. Therefore, this tool is highly effective in meeting said requirements criteria.

Requirement 1.3 is rather essential as the whole purpose behind blockchain is to provide a trusted and valid ledger within a distributed trustless network. If the server application were to accept and share invalid blocks, the project would completely fail at its intended purpose. The solution to this requirement would be to implement validation within the block class implementation itself, and integrate said validation throughout the blockchain server application.

Requirement 1.4 is important for the integrity of the network as a whole. If the complexity of the proof of work puzzle were to be constant, the rate at which new blocks are found would

increase exponentially as the network's user base increases. This could result in new valid blocks being generated before a previously found valid block had fully propagated through the network, which would result in the network 'splitting' its consensus of the blockchain, essentially destroying the network's integrity. A solution for this would be to set a target time for blocks to be generated in, the 'block time', and adjust the proof of work puzzle according to that time in reference to previous blocks generated. If blocks are being produced too fast, the difficulty of the puzzle would increase and vice versa.

Requirements 1.5 to 1.7 are specific to the application implementation itself. In order to satisfy requirement 1.5 and 1.6, Java with the Spring framework is chosen as the language for this project. Java lends itself to requirement 1.5 as the compiled code runs within the JVM that is independent of the operating system. Java's clean and performant multi-threading capabilities with the Runnable interface, and thread pool executors also provides a solution to requirement 1.6. Requirement 1.7 is somewhat innovative providing a solution to one of the main issues surround P2P networks. It is important that peers in the network each have a 'path' to one another so that new blocks are quickly distributed to every node so that consensus on the current state of the blockchain is quickly gained.

The solution for requirement 1.1 is the CryptoNight algorithm that introduces a memory bottleneck within the proof of work. Due to the other requirements involving the use of Java, a CryptoNight solution within such language must be provided, for which the requirement is expanded upon in requirements 2, 2.1 and 2.2. There are two potential solutions for these requirements, a wrapper implementation utilising the Java Native Interface (JNI) with the original CryptoNight binaries, and a 100% pure Java solution. The JNI solution would have the benefit of utilising the all ready optimal and tested algorithm implementation (requirement 2.1), and the Java pure solution would run entirely within the JVM and would be operating system independent (requirement 2.2).

3.2 CryptoNight

The following section details the design, implementation and validation of two Java CryptoNight solutions, CryptoNightJ and CryptoNightJNI, being the first open source Java solutions published. An overview of the algorithms design and how it functions is first given, with diagrams providing a visualisation of each stage in the algorithm (section 3.2.1). A detailed look at how the various mechanisms that make up the algorithm are converted into a pure Java format within the CryptoNightJ solution is then provided (section 3.2.2). Finally, an explanation of the Java Native Interface solution, that provides a wrapper around the original CryptoNight binaries is given (section 3.2.3).

3.2.1 Algorithm Design And Specification

The CryptoNight algorithm consists of three stages: scratchpad initialisation, memory-hard loop and results calculation. Throughout the algorithm, a 200 byte 'keccak state' based on the initial input has various cryptographic functions and transformations applied to it, with the resultant hash produced being of size 256 bits. The following subsections describe each of these stages in detail, with diagrams and pseudocode provided to complement the explanations.

3.2.1.1 Scratchpad Initialisation

The first stage in the algorithm as defined within the CryptoNight specification (Seigen et al., 2013) is the allocation and initialisation of a 2MB scratchpad within memory. As mentioned previously, 2MB is chosen as CPUs at the time of CryptNight's inception generally had 2MB of L3 cache, however the scratchpad size can be increased and adjusted according to modern CPU developments.

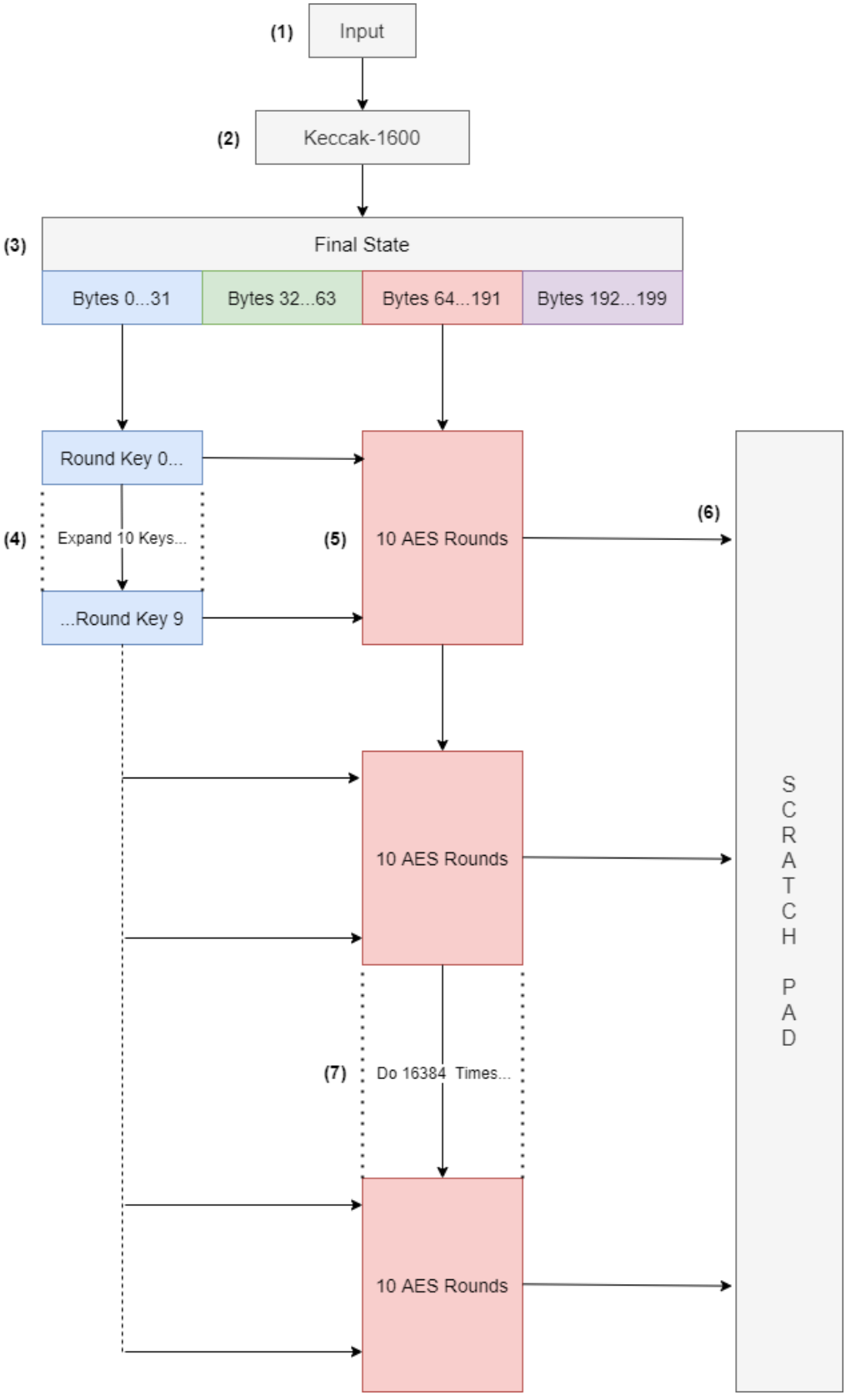


Figure 3.1: CryptoNight scratchpad initialisation stage

Figure 3.1 shows a diagram of how the scratchpad initialisation is carried out. Section (1) represents the hashing algorithms input, which is converted into bytes from its original format, generally through UTF-8 encoding.

The input bytes are then hashed using the `Keccak-1600` algorithm shown in Figure 3.1 section (2). `Keccak` is a family of hashing algorithms that are based on the sponge construction. The sponge construction is term in cryptography that defines functions that operate based on a fixed-length permutation and padding rule, which provides variable-length input to variable-length output mapping (Bertoni et al., 2007). The `Keccak` specification (STMicroelectronics et al., 2008) defines seven permutations, b, of various byte widths including: 25, 50, 100, 200, 400, 800 and 1600 bits. The capacity, c, determines the 'security strength level' of the algorithm as outlined by the NIST SP 800-57 specifications (Barker et al., 2007). The most well known and used `Keccak` implementation is SHA3-256 (Secure Hashing Algorithm), which produces a 256 bit with a capacity of $c = 512$.

The CryptoNight algorithm works on 200 bytes, therefore values of $b = 1600$ and $c = 512$ are used, and the entire 1600 bit state of the hash is passed on and defined as the 'final state' as seen in Figure 3.1 section (3).

The first 32 bytes of the final state is extracted and interpreted as an AES-256 (Advanced Encryption Standard) key and expanded to produce 10 round keys as defined by the AES specification key scheduling algorithm (NIST-FIPS, 2001). The first 16 bytes of the AES-256 key is formatted into a $4 * 4$ matrix with each column of the block denoting a word, w_i (with 4 words per 16 byte block). The format of the 16 byte block is can be seen in Figure 3.2, where K_i denotes the first 16 bytes from the AES-256 key.

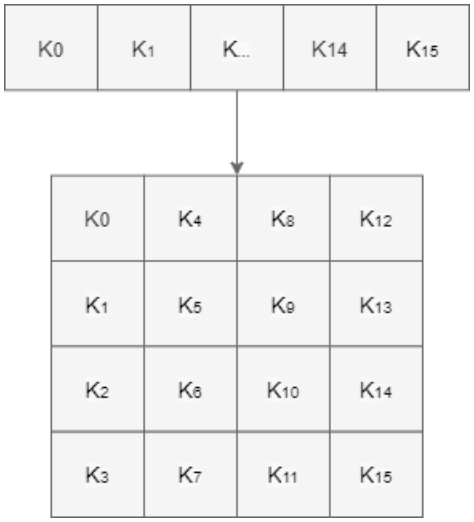


Figure 3.2: 16 byte AES block format

To generate a subsequent word (for the example in Figure 3.2 the next word would be w_5 which would include bytes $K_{16}...K_{19}$), equation Eq. (3.1) is calculated where the function, f, carries out: `RotWord()` (one-byte circular rotation), `SubWord()` (S-box substitution, see Appendix A), and an XOR with the round constant.

$$w_{i+1} = w_i \oplus f(w_{i-1}) \tag{3.1}$$

Since 10 round keys are required, 36 consecutive words are generated using this equation producing 10 keys of 16 bytes, seen in Figure 3.1 section (4). Bytes 64 to 191 (128 bytes total) from the final state are extracted and are split into 8 blocks of 16 bytes each. The `aes_round()` function is then then applied on each block 10 times for each key as shown in Listing 3.1 which corresponds to Figure 3.1 section (5).

```
for i = 0 to 9 do:
    block = aes_round(block, round_keys[i])
```

Listing 3.1: `aes_round()` cycle on a block

The `aes_round()` function is a modified version of the standard AES round, which carries out the `SubBytes()`, `ShiftRows()` and `MixColumns()` transformations on the block before being XORed with the provided round key. The `SubBytes()` method carries out a substitution of each byte in the block using the AES S-box lookup matrix (see Appendix A). The `ShiftRows()` method rearranges each row as shown in Figure 3.3 where $B_0 ... B_{15}$ denotes the 16 bytes of the block.

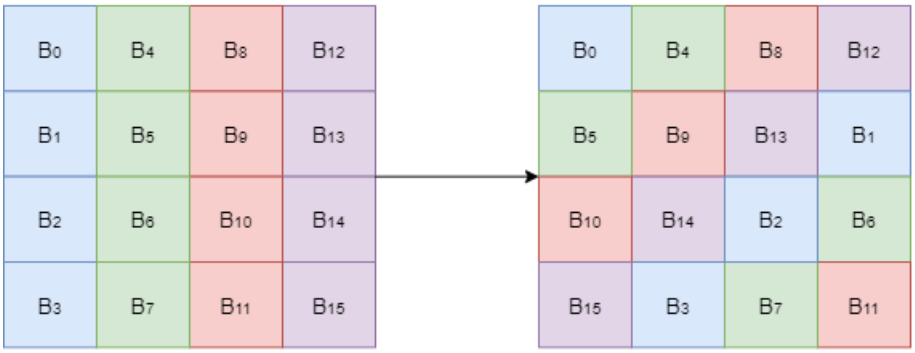


Figure 3.3: `ShiftRows()` transformation

During the `MixColumns()` function, each column is multiple by the fixed matrix shown in Eq. (3.2), where $B_0 ... B_3$ denotes the bytes of a column within the block to be transformed.

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{3.2}$$

The encrypted block is then flattened into a single byte array of length 128, which is written to the first 128 bytes of the scratchpad, seen in Figure 3.1 section (6). An `aes_round()` cycle (see Listing 3.1) with the same round keys as the previous cycle is then carried out on the encrypted block to produce another 128 bytes. This 128 byte block is then written to the next 128 bytes of the scratchpad. This process is iterated 16384 times until the scratchpad has been completely filled with 128 byte `aes_round()` encrypted blocks, seen in Figure 3.1 section (7). With this, the scratchpad is fully initialised.

The full pseudocode for the scratchpad initialisation stage can be seen in Listing 3.2 which correlates to the diagram in Figure 3.1.

```
# Keccak-1600 hash input and assign 2MB scratchpad
finalstate = keccak1600(input)
scratchpad = byte[2097152]

# Expand first 32 bytes of final state to produce 10 round keys
expandedKey = aesKeySchedual(finalstate[0:32])
for i = 0 to 9 do:
    roundkeys[i] = expandedKey[i*16:i*16 + 16]

# Split final state bytes 64...191 into 8x16 blocks
for i = 0 to 7 do:
    blocks[i] = final_state[64:64 + i*16]

# Fill the scratchpad with AES encrypted 128 byte blocks
for i = 0 to 16384 do:
    for j = 0 to 7 do:
        for k = 0 to 9 do:
            blocks[j] = aes_round(blocks[j], round_keys[k])
        scratchpad[i*128:i*128 + j*16] = blocks[j]
```

Listing 3.2: Scratchpad initialisation full pseudocode

3.2.1.2 Memory-hard Loop

The next stage in the algorithm is the memory-hard loop. As its name suggests, this section of the algorithm makes constant read/write calls to the allocated 2MB scratchpad memory, as well as having various cryptographic functions weaved in-between. This stage is where most of the time is spent within the algorithm, and is really what defines CryptoNight. As mentioned, all though GPGPUs and ASICS would be considerably more effective than CPUs at processing the various cryptographic hashing calculations within this stage, the fact that constant read/write calls are made to memory in between said calculations introduces a bottleneck, for which CPUs would more easily overcome due to having a fast cache.

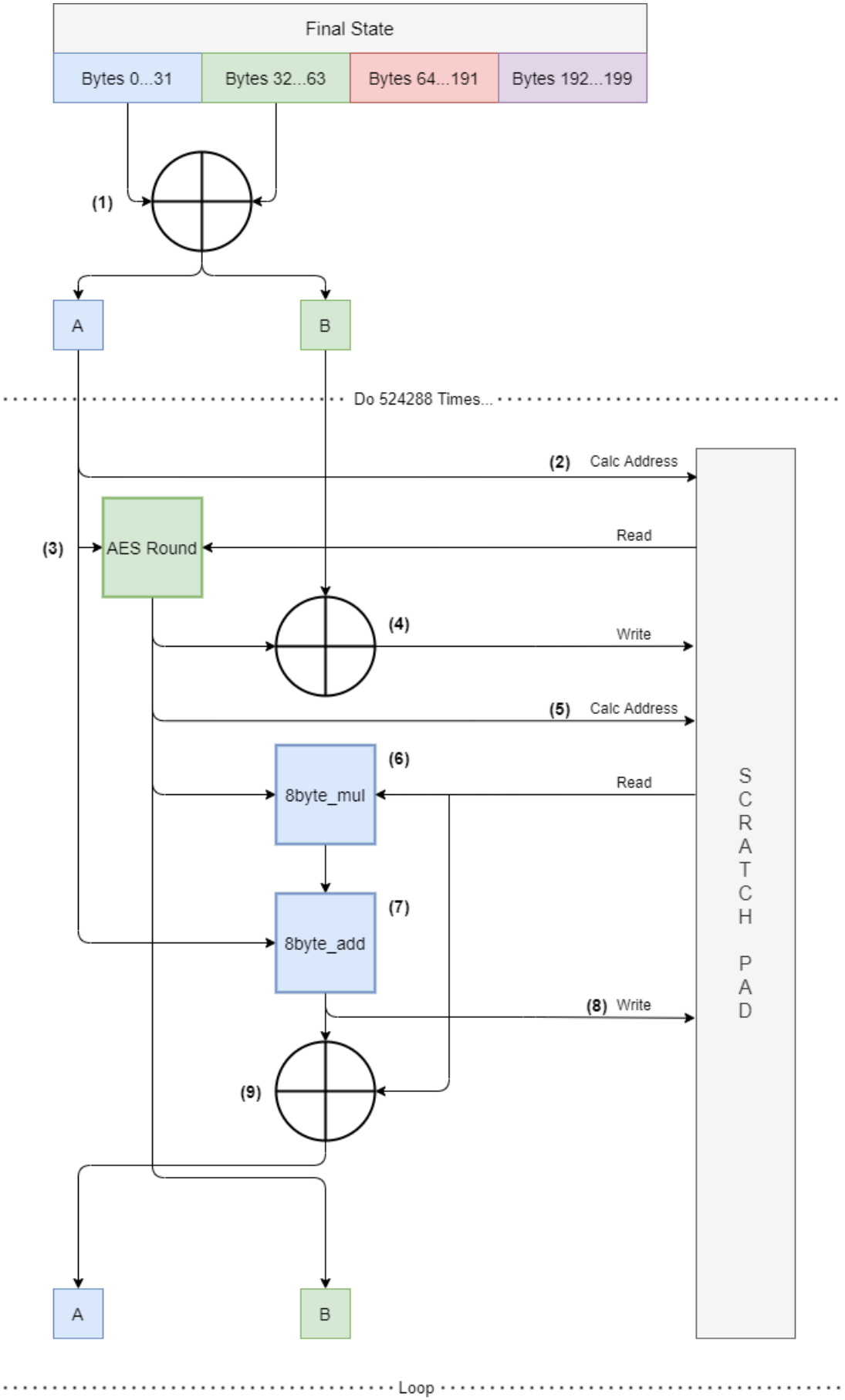


Figure 3.4: CryptoNight memory-hard loop stage

Figure 3.4 shows a diagram of how the memory hard loop is carried out, which is based on the pseudocode provided in the CryptoNight standard (2013) shown in Listing 3.3.

```
scratchpad_address = to_scratchpad_address(a)
scratchpad[scratchpad_address] = aes_round(scratchpad
  [scratchpad_address], a)

b, scratchpad[scratchpad_address] = scratchpad[scratchpad_address],
  b xor scratchpad[scratchpad_address]

scratchpad_address = to_scratchpad_address(b)
a = 8byte_add(a, 8byte_mul(b, scratchpad[scratchpad_address]))

a, scratchpad[scratchpad_address] = a xor
  scratchpad[scratchpad_address], a
```

Listing 3.3: Memory-hard loop pseudocode

The first part of the memory-hard loop shown in Figure 3.4 section (1) is the initialisation of the variables a and b. This is done by XORing the first 32 bytes of the final state with bytes 32 to 63 of the final state, where the first 16 bytes of the result is assigned to a, and the last 16 bytes being assigned to b.

Figure 3.4 section (2) shows the scratchpad address calculation using the `to_scratchpad_address()` method, which is initially calculated using the 16 bytes from a. The `to_scratchpad_address()` method interprets the 16 byte input argument as a little-endian denary value using the 21 low-order bits as the byte index. In order to keep the 16 byte alignment, 4 low-order bits from the byte index are dropped. The value produced is used to index the first byte of the scratchpad to be read, with the 15 bytes after said index position being subsequently read, resulting in 16 bytes from the scratchpad which is returned.

Figure 3.4 section (3) shows the next step in the memory-hard loop, where the 16 bytes read of the scratchpad using the address calculated in section (2) is passed through a single `aes_round()` cycle (transformations shown in Figure 3.3 and Eq. (3.2)), with a being used as the key to be XORed.

The `aes_round()` encrypted 16 byte block is then XORed with the initial 16 bytes from b, which is subsequently written to the scratchpad from the previously calculated scratchpad address, shown in Figure 3.4 section (4). Henceforth, the `aes_round()` encrypted block is assigned to b to be used in subsequent operations, and as the initial value of b in the next iteration.

A new scratchpad address is calculated using the `to_scratchpad_address()` with the new value of b as its input argument, as shown in Figure 3.4 section (5).

16 bytes are read from the scratchpad using the new scratchpad address, which are subsequently passed through the `8byte_mul()` function with the new value of b, as shown in Figure 3.4 section (6). The `8byte_mul()` takes the first 8 bytes of each each input argument and interprets them as unsigned 64 bit little-endian denary values. The values are multiplied together and the result is converted into 16 bytes, for which the first and second half's of the 16 byte array are swapped. The pseudocode for the `8byte_mul()` can be seen in Listing 3.4.

```
8byte_mul(a, b):
    mul = int(a[0:8]) * int(b[0:8])
    res = mul[8:16] + mul[0:8]

    return res
```

Listing 3.4: 8byte_mul() pseudocode

The result from the 8byte_mul() calculation made in Figure 3.4 section (6) is then passed through the 8byte_add() function with the initial value of a, shown in section (7). The 8byte_add() function interprets each input argument as a pair of 64 bit little-endian denary values, for which each respective component is subsequently added by modulo 2^64. There is a chance the resultant value calculated might overflow, which is to be expected. The result is converted into 16 bytes before being returned. The pseudocode for the 8byte_add() can be seen in Listing 3.5.

```
8byte_add(a, b):
    a1, a2 = a[0:8], a[8:16]
    b1, b2 = b[0:8], b[8:16]

    first_add = int(a1+b1)
    second_add = int(a2+b2)

    res = first_add[0:8] + second_add[0:8]

    return res
```

Listing 3.5: 8byte_add() pseudocode

The result of the previously calculated 8byte_add() function is XORed with 16 bytes from the scratchpad using the previously calculated scratchpad address (based of to_scratchpad_address(b)) which is assigned to a to be used in the next iteration. The result of the 8byte_add() function is also written to the scratchpad (at the same scratchpad address) before the current iteration concludes and the next memory-hard loop cycle commences.

3.2.1.3 Results Calculation

The final stage in the CryptoNight algorithm is the results calculation. During this stage, the final state is modified in combination with the scratchpad, resulting in 200 bytes referred to as the modified state. This is required as otherwise the memory-hard loop stage could be bypassed, making said stage redundant, and the algorithm as a whole ineffective towards its intended purpose. The modified state is passed through a final Keccak-1600 permutation before having one of the following hashing functions selected based on said permutation: BLAKE-256, Groestl-256, JH-256 and Skein-256. The selected hash is then applied to the modified state permutation which produces the CryptoNight Hash.

Figure 3.5 details the process that is carried out in order to produce the final CryptoNight hash. First, bytes 64 to 191 from the final state are extracted and are XORed with the first 128 bytes of the scratchpad. Bytes 32 to 63 are extracted from the final state and is interpreted as an AES-256 key, which is subsequently expanded to produce 10 round keys (as was done during the scratchpad initialisation stage), shown in Figure 3.5 section (2).

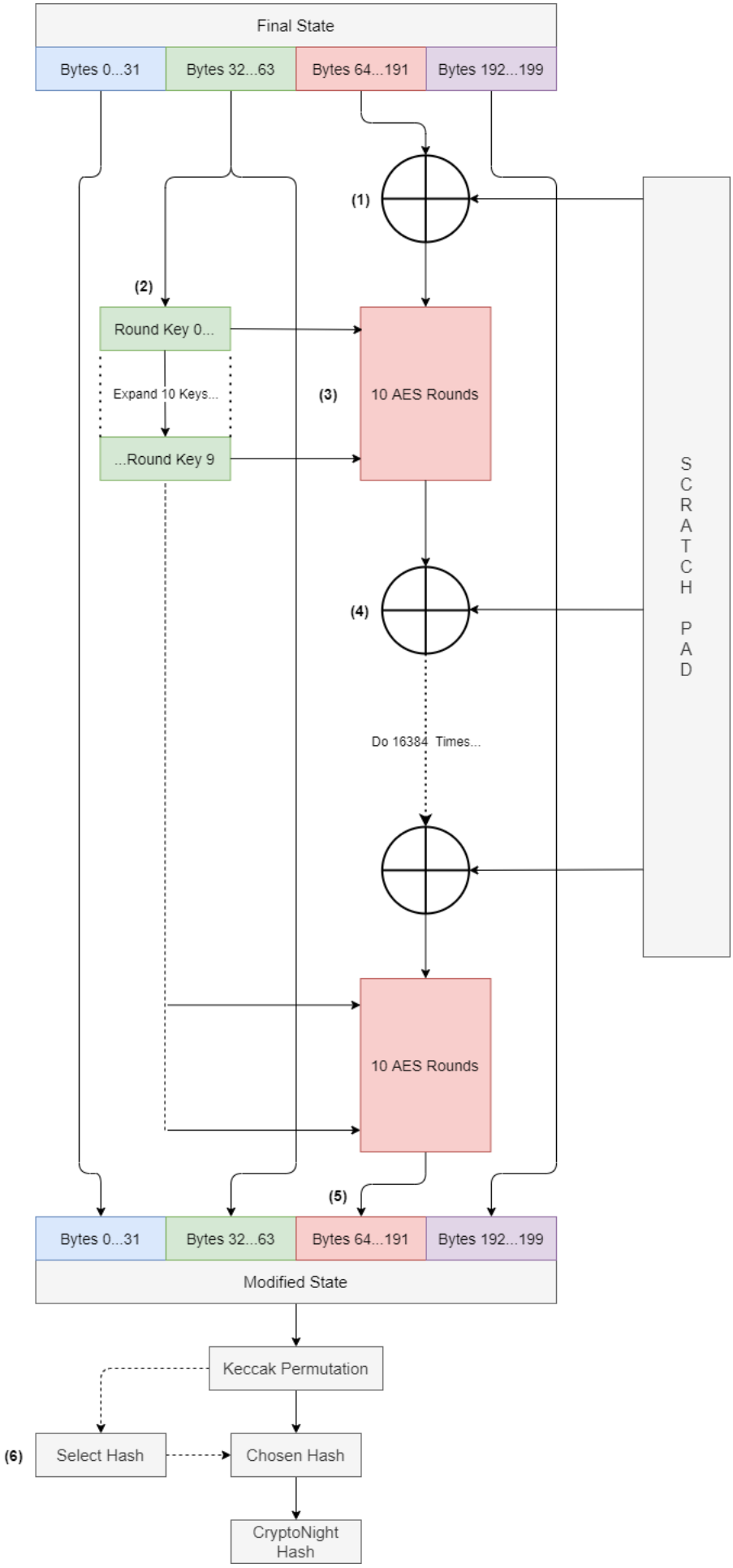


Figure 3.5: CryptoNight results calculation stage

The results of the XOR calculated in section (1) of the diagram is then passed through a full 10 key cycle of the `aes_round()` method (see Listing 3.1) using the keys produced in section (2). The 128 byte AES encrypted block is then XORed with the next 128 of the scratchpad, which goes through a subsequent `aes_round()` cycle, and so on. This process is iterated 16384 times until each 128 byte section of the scratchpad has been XORed with a corresponding AES encrypted block.

Once the final 128 bytes of the scratchpad has been XORed, a final `aes_round()` cycle is carried out on the 128 byte block, before its written to bytes 64 to 191 of the final state to produce the modified state, shown in Figure 3.5 section (5).

The modified state is then passed through a `Keccak-1600` permutation, before the final hash is selected based on the two low-order bits. The selection process based on said bits is as follows: 0=BLAKE-256, 1=Groestl-256, 2=JH-256 and 3=Skein-256.

The hashing algorithms used for the final hash are the four SHA3 candidate finalists that ultimately lost to Keccak during the SHA3 selection competition (Chang et al., 2012). They each produce a 256 bit output that is used as the CryptoNight hash. The full pseudocode for the results calculation stage can be seen in Listing 3.6 which correlates to the diagram in Figure 3.5.

```
# Expand bytes 64 to 191 of final state to produce 10 round keys
expandedKey = aesKeySchedual(finalstate[0:31])
for i = 0 to 9 do:
    roundkeys[i] = expandedKey[i*16:i*16 + 16]

# Extract bytes 64 to 191 of the final state
aesBlock = final_state[64:191]

for i = 0 to 16384 do:
    # XOR the 128 byte AES block with 128 bytes from scratchpad
    aesBlock = xor(aesBlock, scratchpad[i*128:i*128 + 128])
    for j = 0 to 7 do:
        blocks[j] = aesBlock[64:64 + j*16]

    # AES round encrypt and store new block for next iteration
    for j = 0 to 7 do:
        for k = 0 to 9 do:
            blocks[j] = aes_round(blocks[j], round_keys[k])
        aesBlock[j*16:j*16 + 16] = blocks[j]

# Replace bytes 64 to 191 of the final state with the 128 byte aes block
final_state[64:191] = aesBlock

# Pass final state through a keccak 1600 permutation
modified_state = keccak1600(final_state)

# Select final hashing function based of modified state 4 low-order bits
selector = modified_state[0] & 3
switch(selector):
    0: res = blake_256(modified_state)
    1: res = groestil_256(modified_state)
    2: res = jh_256(modified_state)
    3: res = skein_256(modified_state)

return res
```

Listing 3.6: Scratchpad initialisation full pseudocode

3.2.2 Pure Java Solution (CryptoNightJ)

The first Java solution for the CryptoNight algorithm is the native implementation that relies on no external non Java components, unlike the JNI solution. The project has a single module, JCryptoNight, which is built and structured using maven. The full source code can be in the repository located at: `https://github.com/jounaidr/CryptoNightJ`.

3.2.2.1 Class Dependencies and Project Structure

Figure 3.6 shows the class structure overview for the CryptoNightJ project, with a full package expanded version provided in Appendix B. The UML class diagrams can also be seen in Appendix C.

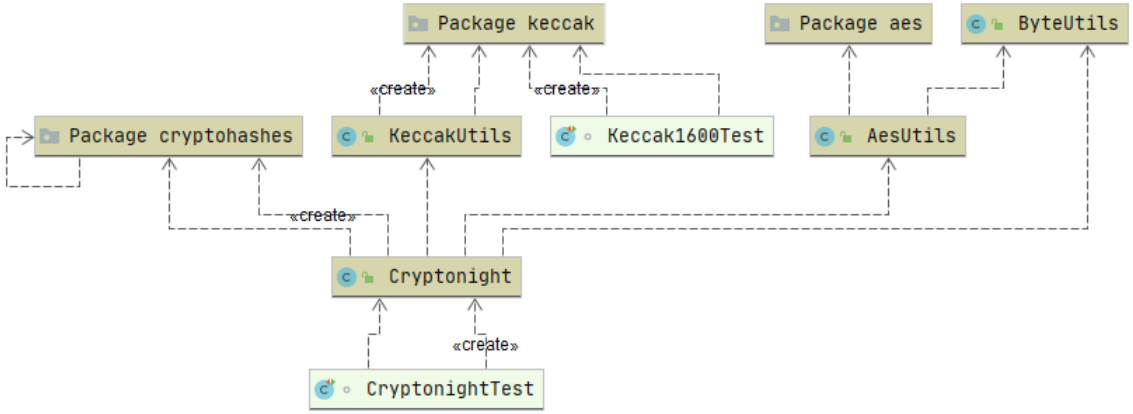


Figure 3.6: CryptoNightJ class/package dependency overview diagram

The main class that carries out the: scratchpad initialisation; memory-hard loop and results calculation stages of the CryptoNight algorithm is `Cryptonight.java`. The class is initialised with the a string constructor argument `inputData` which denotes the input string to be hashed. The CryptoNight algorithm stages are carried out in the constructor, with the private class variable `out` being assigned the resultant hash upon completion of the algorithm, which can be returned using the `returnHash()` public method.

The `KeccakUtils` class contains helper methods around the keccak hashing function. The first method, `keccak1600()`, is used to do the initial `Keccak-1600` hashing which utilises a modified version of Melsha's keccak solution (found in the `Keccak` package). The second method, `permutation()`, carries out a single keccak 1600 permutation, which is adapted from Yi-Cyuan's JavaScript keccak solution.

The `AesUtils` class contains various helper methods to carry out the AES key expansion and encryption. The methods relating to key expansion are taken from the cryptoexplained solution (with classes in the `aes` package). The `subBytes()`, `shiftRows()` and `mixColumns()` methods are taken from Chen's solution and are implemented in the `aesRound()` method shown in Listing 3.7 (based on the design outlined in Listing 3.1, Figure 3.3 and Eq. (3.2)).

```
1  public static byte[] aesRound(byte[] in, byte[] key){
2      byte[][] state = new byte[4][4];
3
4      // Convert input into into 4x4 block in AES format
5      for (int i = 0; i < state.length; i++){
6          for (int j = 0; j < state[i].length; j++){
7              byte[] portion = Arrays.copyOfRange(in, 4*i, 4*i+4);
8              state[j][i] = portion[j];
9          }
10     }
11
12     // Carry out AES methods
13     state = subBytes(state);
14     state = shiftRows(state);
15     state = mixColumns(state);
16
17     // Flatten block into 16 byte array
18     byte[] out = new byte[16];
19     for (int i = 0; i < state.length; i++){
20         for (int j = 0; j < state[i].length; j++){
21             out[4*i + j] = state[j][i];
22         }
23     }
24
25     // XOR the output with the provided round key
26     out = ByteUtils.xor(out, key);
27     return out;
28 }
```

Listing 3.7: `aes_round` implementation in AesUtils.java

The `aesRound()` method takes in two byte array arguments for the block input and round key, both of size 16. Lines 5 to 10 consist of a nested for loop that splits the 16 byte input array into a $4 * 4$ 2D array. The `Arrays.copyOfRange()` utility method is used to extract single byte sections from the array, which are subsequently placed in their respective position in the 2D array (line 8). Note that the i and j position of where the byte is set is the reverse of the nested for loop iterators, which places the bytes by column order rather than row order to comply with the AES block structure (shown in Figures 3.2 and 3.3). The `subBytes()`, `shiftRows()` and `mixColumns()` transformations are then applied to the formatted block (lines 13 to 15) before the block array is flattened back into a single dimension. The `key` input argument is then XORed with the flattened block which completes the CryptoNight special case `aes_round` method described in section 3.2.1.1 of the algorithm design.

The `ByteUtils` class contains various helper methods around byte array which are used in both the main `Cryptonight` class as well as the `AesUtils` class. The main function from this utility class that is frequently used throughout the code is the `xor()` method shown in Listing 3.8.

```
1  public static byte[] xor(byte[] in1, byte[] in2) {
2      byte[] out = new byte[Math.min(in1.length, in2.length)];
3      for (int i = 0; i < out.length; i++) {
4          out[i] = (byte) ((in1[i] ^ in2[i]) & 0xFF);
5      }
6      return out;
7  }
```

Listing 3.8: `xor()` helper method in ByteUtils.java

There are two reasons why this method is required to XOR byte arrays in Java. First, each respective byte component of the byte arrays must be XORed individually, requiring iteration through said byte arrays (line 3). The second reason is that all Java byte data types are signed and therefore the `0xFF` mask must be applied to each byte in the output array to produce correct a result (line 4). A full list of methods contained within the `ByteUtils` utility class can be seen in its respective UML class descriptor in Appendix C.

The four hashing functions (BLAKE-256, Groestl-256, JH-256 and Skein-256) required during the results calculation stage of the CryptoNight algorithm can be found in the `cryptohashes` package, which utilises classes from Fuhrmann (2014)'s Saphir Hash Java cryptography library. The library implements said functions using the `Digest` interface as per the JDK and Bouncy Castle (2000) Java crytography standards/framework. The structure of these implementations and how they are integrated into the `Crytonight` class can be seen in Appendix B.

3.2.2.2 Scratchpad Initialisation Java Implementation

The first part of the algorithm as discussed in the CryptoNight design section 3.2.1.1 is the scratchpad initialisation. The psudocode shown in Listing 3.2 is adapted into Java utilising the discussed helper methods, and is shown in Listing 3.9.

```
1  //*****************************************************************
2  //                    SCRATCHPAD INITIALISATION
3  //*****************************************************************
4
5  //Initialise 2mb scratch pad
6  byte[] scratchPad = new byte[2097152];
7  //Hash input data with Keccak-1600 to produce 200 byte 'final state'
8  byte[] finalState = KeccakUtils.keccak1600(inputData);
9
10 //Extract first 32 bytes of the final state and expand to produce 10 16
       byte round keys,
11 //and format round keys as a 10x16 2D array
12 byte[] expandedKeys = AesUtils.expandRoundKeys(Arrays.copyOfRange(
       finalState, 0, 32),10);
13 byte[][] keys = new byte[10][16];
14 for (int i = 0; i < 10; ++i){
15     System.arraycopy(expandedKeys, 16 * i, keys[i], 0, 16);
16 }
17
18 //Extract bytes 64 to 191 of the final state and format into 8 16 byte
       blocks (as 2D array)
19 byte[][] blocks = new byte[8][16];
20 for (int i = 0; i < 8; ++i){
21     System.arraycopy(finalState, 64 + 16 * i, blocks[i], 0, 16);
22 }
23
24 //Encrypt blocks over and over again with 'aes_round' (with 10 keys
       generated previously),
25 //and write each 128 bytes produced to the scratchpad consecutively,
26 //where 'aes_round' carries out SubBytes, ShiftRows and MixColumns steps
       from AES specification
27 for(int i = 0; i < 16384; i++){
28     //For each of the 8 blocks, apply 'aes_round' to each block with
29     //the 10 generated round keys
30     for (int j = 0; j < 8; ++j){
31         for (int k = 0; k < 10; ++k) {
32             blocks[j] = AesUtils.aesRound(blocks[j], keys[k]);
33         }
34         //Write the encrypted blocks to the 128 bytes in the
35         //scratch pad from position i
36         System.arraycopy(blocks[j], 0, scratchPad, i*128 + j*16, 16);
37     }
38 }
```

Listing 3.9: Scratchpad initialisation in Cryptonight.java

First, the scratchpad byte array is assigned (line 6) of size 2MB (2097152 length byte array). The final state is then calculated from the input data using the `KeccakUtils.keccak1600()` utility method to produce the 200 byte `finalState` array (line 8). The `AesUtils.expandRoundKeys()` method is used to produce the 160 byte `expandedKeys` array, using the 32 bytes of the final state extracted using `Arrays.copyOfRange()` (line 12). The `expandedKeys` array is then formatted into the $10 * 16$ 2D `keys` array in lines 13 to 16.

The 128 bytes section of the final state (from bytes 64 to 191) is extracted into the $8 * 16$ `blocks` array using `System.arraycopy()` in lines 19 to 22.

The `blocks` array is then put though 16,384 10 key `AesUtils.aesRound()` encryption cycles (1,310,720 encryption rounds in total) with the 128 bytes produced in each cycle being consecutively written to the `scratchPad` array (lines 27 to 38). This is done using three nested for loops that: iterate through each 128 byte section of the scratchpad (line 27); iterate through each 16 byte block of the 128 byte AES encrypted array (line 30) and iterate through each round key in the `keys` array to carry out the `AesUtils.aesRound()` cycles. Once said iterations have concluded, the scratchpad is fully initialised and ready to be used in the next stages of the algorithm.

3.2.2.3 Memory-hard Loop Java Implementation

As mentioned in the algorithm design section, the memory-hard loop is the most complex and computationally expensive stage in the CryptoNight algorithm. Read/write calls are made to the scratch in a specific order, which if carried out incorrectly will result in vastly a different scratchpad than what is required, producing an incorrect hash in the later stages. The pseudocode provided in the CryptoNight specification (Seigen et al., 2013) (shown in Listing 3.3) is addapted into Java, shown in Listing 3.10.

```
//*****************************************************************
//                          MEMORY-HARD LOOP
//*****************************************************************

//Extract bytes 0 to 31 as 'finalState1', and bytes 32 to 63 as
//'finalState2', and xor them, with the first 16 bytes of the
//result being assigned to 'a' and the rest assigned to 'b'
byte[] finalState1 = Arrays.copyOfRange(finalState, 0, 32);
byte[] finalState2 = Arrays.copyOfRange(finalState, 32, 64);
byte[] abXor = ByteUtils.xor(finalState1, finalState2);
byte[] a = Arrays.copyOfRange(abXor, 0, 16);
byte[] b = Arrays.copyOfRange(abXor, 16, 32);

for (int i = 0; i < 524288; ++i){
    //Interoperate a scratchpad address using 'a' 16 bytes
    int scratchpadAddress = toScratchpadAddress(a);

    //Use scratchpad address to read 16 bytes from scratchpad,
    //and carry out one round of the 'aes_round' method
    //using 'a' as the key
    byte[]scratchpadAes = AesUtils.aesRound(Arrays.copyOfRange(scratchPad,
     scratchpadAddress, scratchpadAddress + 16), a);

    //Xor 'b' and the previously calculated 'scratchpad_aes' 16 bytes,
    //and write result to scratchpad at the 'scratchpadAddress',
    //before assigning 'scratchpad_aes' to 'b'
    byte[] xorB = ByteUtils.xor(b, scratchpadAes);
    System.arraycopy(xorB, 0, scratchPad, scratchpadAddress, 16);
    b = scratchpadAes;

    //Interoperate a scratchpad address using 'b' 16 bytes
    scratchpadAddress = toScratchpadAddress(b);

    //Calculate '8byte_mul' of 'b' and 8 bytes of the scratchpad from
    //'scratchpadAddress', and calculate 'f8byteAdd' on the result of
    //the previous calculation and 'a'. Assign the result to 'a'
    a = f8byteAdd(a, f8byteMul(b, Arrays.copyOfRange(scratchPad,
    scratchpadAddress, scratchpadAddress + 8)));

    //Finally, calculate the xor of 'a' and 16 bytes of scratchpad
    //from 'scratchpadAddress'. Write the 16 bytes of 'a' (before xor)
    //to the scratchpad at 'scratchpadAddress', before assigning the
    //previously calculated xor to 'a' to complete the iteration
    byte[] xorA = ByteUtils.xor(a, Arrays.copyOfRange(scratchPad,
    scratchpadAddress, scratchpadAddress + 16));
    System.arraycopy(a, 0, scratchPad, scratchpadAddress, 16);
    a = xorA;
}
```

Listing 3.10: Memory-hard loop in Cryptonight.java

The first part of this stage consists of the initialisation of the various variables used within memory-hard loop. The two sections of the final state that are required as specified in Figure 3.4 section (1) are extracted into the byte arrays `finalState1` and `finalState2` using `Arrays.copyOfRange()` (lines 8 and 9). A subsequent XOR of these byte arrays using the `ByteUtils.xor()` utility method is split into two 16 byte components and assigned to byte arrays `a` and `b` respectively (lines 10 to 12).

After said variables have been initialised, the memory-hard loop commences with 524,288 iterations (line 14). As detailed in the algorithm design section 3.2.1.2, the 16 bytes of `a` are interpreted as a scratchpad address integer using the `toScratchpadAddress()` method (line 16) for which the contains of said method can be seen in Listing 3.11.

```
1 private int toScratchpadAddress(byte[] a) {
2     return  ((a[0] & 0xFF) >> 4) << 4 |
3             ((a[1] & 0xFF) << 8) |
4             (((a[2] & 0xFF) & 0x1F) << 16);
5 }
```

Listing 3.11: Cryptonight.java `toScratchpadAddress()` method

The `toScratchpadAddress()` implementation works on the first three bytes of the provided byte array input argument, and performers various bitwise transformations on them with the purpose of providing a correctly formatted scratchpad address, which is subsequently returned (as an integer datatype).

The `scratchpadAddress` is then used to read 16 bytes from the scratchpad (using the 15 subsequent bytes from the one at the specified address) using the `Arrays.copyOfRange()` method, and is passed through a single `AesUtils.aesRound()` cycle with the 16 bytes from `a` as the key.

The resultant 16 bytes is stored in the temporary byte array, `scratchpadAes`, which is subsequently XORed with `b` (line 26), with the 16 bytes retuned being written to the scratchpad at the `scratchpadAddress` (line 27). After the scratchpad write is complete, `b` is assigned the 16 bytes from `scratchpadAes` which is used during the next iteration as outlined in Figure 3.4.

A new `scratchpadAddress` is calculated using the 16 bytes from `b` (line 31), which is used to read 8 bytes from the scratchpad which passed into the `f8ByteMul()` function along with `b`. The resultant 16 bytes is subsequently passed into the `f8ByteAdd()` function along with `a`, for which the resultant 16 bytes is assigned to `a` (line 36). The `f8ByteMul()` and `f8ByteAdd()` implementations are adapted from Ilia's JavaScript solution.

Finally, the XOR of `a` and 16 bytes from the scratchpad at `scratchpadAddress` is calculated, before writing the 16 bytes of `a` to the scratchpad at the `scratchpadAddress` using the `Arrays.copyOfRange()` method. The previous XOR result is then assigned to `a`, which is used in the next iteration as outlined in Figure 3.4.

With this, a single iteration of the memory-hard loop is complete, with 524,287 additional iterations being carried out until the memory-hard loop has concluded and the scratchpad is ready to be applied to the final state.

3.2.2.4 Results Calculation Java Implementation

The final stage of the algorithm as discussed in the CryptoNight design section 3.2.1.3 is the results calculation. The stage is relatively simple in comparison to the previous, and utilises many of the same functions and processes of the scratchpad initialisation stage. The psudocode shown in Listing 3.6 is adapted into Java utilising the discussed helper methods, and is shown in Listing 3.12.

```
//*****************************************************************
//                          RESULTS CALCULATION
//*****************************************************************

//Generate 10 AES round keys from bytes 32 to 63 (32 bytes) in the
//same manner as was previously done during scratchpad initialisation
expandedKeys = AesUtils.expandRoundKeys(Arrays.copyOfRange(finalState, 32,
    64),10);
for (int i = 0; i < 10; ++i){
    System.arraycopy(expandedKeys, 16 * i, keys[i], 0, 16);
}

//Read 128 byes from bytes 64 to 191 of the scratchpad which will be used
//to consecutively xor with each 128 byte section of the scratchpad
byte[] keccakXorAes = Arrays.copyOfRange(finalState, 64, 192);

for(int i = 0; i < 16384; i++) {
    //xor the current 128 'keccakXorAes' bytes from the final state
    //with 128 bytes of the scratchpad from address 'i'
    //(16384 denotes the number of 128 byte blocks in the scratchpad)
    keccakXorAes = ByteUtils.xor(keccakXorAes, Arrays.copyOfRange(
   scratchPad, i * 128, (i * 128) + 128));

    //Split the 128 bytes into 8 16 byte blocks
    for (int j = 0; j < 8; ++j){
        System.arraycopy(keccakXorAes, 16 * j, blocks[j], 0, 16);
    }

    //Carry out a full cycle of 'aes_rounds' methods with the
    //10 AES keys expanded previously, and write the encrypted blocks back
    //into the 128 byte 'keccakXorAes' to be used in the next iteration
    for (int j = 0; j < 8; ++j){
        for (int k = 0; k < 10; ++k) {
            blocks[j] = AesUtils.aesRound(blocks[j], keys[k]);
        }
        System.arraycopy(blocks[j], 0, keccakXorAes, j*16, 16);
    }
}

//Write the modified 128 bytes back into the
//final state at bytes 64 to 191
System.arraycopy(keccakXorAes, 0, finalState, 64, 128);

//Pass the final state through the 'Keccak-f' function that carries out a
//keccak permutation, and calculate the result hash to be used from the
//first 2 bits of the modified state
byte[] modifiedState = KeccakUtils.permutation(finalState);
int lastHashType = (modifiedState[0] & 0xFF) & 3;

//Hash the modified state with the chosen hash function which produces the
//cryptonight hash assigned to the global 'out' byte array.
```

```
50 switch(lastHashType) {
51     case 0: // BLAKE-256
52         BLAKE256 blake256 = new BLAKE256();
53         out = blake256.digest(modifiedState);
54         break;
55     case 1: // GROESTL-256
56         Groestl256 groestl256 = new Groestl256();
57         out = groestl256.digest(modifiedState);
58         break;
59     case 2: // JH-256
60         JH256 jh256 = new JH256();
61         out = jh256.digest(modifiedState);
62         break;
63     case 3: // SKEIN-256
64         Skein256 skein256 = new Skein256();
65         out = skein256.digest(modifiedState);
66         break;
67 }
```

Listing 3.12: Results calculation in Cryptonight.java

Similar to scratchpad initialisation, the first step involves AES key expansion to generate 10 16 byte round keys, for which the initial AES-256 key is extracted from the final state bytes 32 to 63, using `Arrays.copyOfRange()` (line 7). The previously declared `keys` array (from scratchpad initialisation 3.2.2.2) is reused and is populated in the same way as was done previously (lines 8 to 10).

Before the final state modification loop commences, bytes 64 to 191 of the final state are stored in the `keccakXorAes` byte array (line 14). This 128 byte section is what is subsequently modified using the scratchpad which introduces a dependency between the scratchpad and final hash, as is shown in Figure 3.5.

The final state section modification first starts with an XOR between the current 128 byte block stored in `keccakXorAes`, and the current the 128 byte section of the scratchpad that is determined by the loop iterator `i` (line 20).

The 128 bytes is subsequently split into blocks of 16 bytes reusing the previously declared `blocks` 2D array (lines 23 to 25), before each block is passed through a full 10 key `AesUtils.aesRound()` encryption cycle, with the resultant encrypted blocks being consecutively written to the `keccakXorAes` variable to be used in the next iteration (lines 30 to 35).

This processes is iterated 16384 times until each 128 byte section of the scratchpad has been XORed with the `keccakXorAes` array. Note that each iteration uses the same round keys generated in the first step of this stage.

After the loop has concluded, the resultant 128 bytes from `keccakXorAes` are placed back into the final state section it was taken from (line 40) using `System.arraycopy()`, which produces the modified state. The modified state is subsequently passed through a keccak 1600 permutation using the `KeccakUtils.permutation()` method (line 45). The result is then used to determine the `lastHashType` by calculating the AND bitwise operation of the first byte in the modified state, and 3 (11 in binary), which produces a integer between 0 and 3 (line 46).

A switch statement is then used to apply the final hash based on the `lastHashType` to the modified state as outlined in the algorithm design section 3.2.1.3, with the four hashing functions required implemented using from Fuhrmann's cryptography library (lines 50 to 66).

The resultant CryptoNight hash is stored in the `out` class variable as previously mentioned, which can be accsessed from outside the class using the `returnHash()` public method.

3.2.2.5 CryptoNightJ Testing and Validation

There are two unit tests written using JUnit associated with Cryptonight.java (for which the test class dependency can be seen in Figure 3.6), `TestHashCorrect()`, which tests for the algorithms validity, and `TestHashSpeed()`, which tests the algorithms performance. The test cases use five known hashes which can be seen in Table 3.1, for which the contents of both unit test methods can be seen in Listing 3.13.

Table 3.1: Valid hashes used in `TestHashCorrect()`

No.	Input String	Hash
1	"This is a test"	a084f01d1437a09c6985401b60d43554ae105802c5f5d8a9b3253649c0be6605
2	""	eb14e8a833fac6fe9a43b57b336789c46ffe93f2868452240720607b14387e11
3	"de omnibus dubitandum"	2f8e3df40bd11f9ac90c743ca8e32bb391da4fb98612aa3b6cdc639ee00b31f5
4	"caveat emptor"	bbec2cacf69866a8e740380fe7b818fc78f8571221742d729d9d02d7f8989b87
5	"ex nihilo nihil fit"	b1257de4efc5ce28c6b40ceb1c6c8f812a64634eb3e81c5220bee9b2b76a6f05

Hashes No.1 and No.2 are the two valid hashes provided in the CryptoNight specifiction document (Seigen et al., 2013), which both use Groestl-256 as their final hashing function. Hashes No.3 to No.5 are taken from the Monero source code (2014) and use BLAKE-256, JH-256 and Skein-256 as their last hashing functions respectively. The `TestHashCorrect()` test simply instantiates a `Cryptonight.java` object for each of the input strings listed in Table 3.1, and carries out an `assertEquals()` with the generated hash against the valid hash. This test case is crucial during the development and further optimisations of the algorithm as it allows for confirmation that changes made to the code still produce valid hashes. The `TestHashSpeed()` test calculates each of the hashes listed in Table 3.1 1000 times whilst recording execution time data. The data is then used to calculate the hash rate of the algorithm using the equation Eq. (3.3) with hash rate h and total execution time t.

$$h = 5000 \div \frac{t}{1000} \tag{3.3}$$

```
@Test
public void TestHashCorrect(){
    for (int i = 0; i < inputData.size(); i++) {
        Cryptonight cryptonight = new Cryptonight(inputData.get(i));
        assertEquals(validHashes.get(i),new String(Hex.encode(cryptonight
    .returnHash())));
    }
}

@Test
public void TestHashSpeed(){
    long totalTime = 0;
    for (String inputData : inputData) {
        for (int x = 0; x < 1000; x++) {
            long startTime = System.currentTimeMillis(); //start timer
            Cryptonight cryptonight = new Cryptonight(inputData);
            long endTime = System.currentTimeMillis();
            totalTime = totalTime + (endTime - startTime); //end timer
        }
    }
    float hashRate = (5000 / ((float)totalTime / 1000));
}
```

Listing 3.13: Cryptonight.java unit tests

3.2.3 JNI Wrapper Solution (CryptoNightJNI)

The second Java solution for the CryptoNight algorithm involves the use of the Java Native Interface (JNI) to execute binaries of the original C++ implementation of said algorithm. As outlined by Stepanian et al. (2005), the JNI provides an API that allows for the interoperability of non-Java code within a Java application. It works at the function level with direct mappings between the Java and non-Java methods, with the Java methods having the ability to call the equivalent methods within external code (referred to as 'callouts'). The external code also has the ability to access and modify data within the Java code through the JVM, which allows for the return of data to be converted into Java datatype formats (referred to as 'callbacks').

The CryptoNightJNI source code can be found at: `https://github.com/jounaidr/CryptoNightJNI`, with the artifacts also available on the Maven Central Repository (see section 3.2.3.3 for more detail).

3.2.3.1 CryptoNightJNI Overview and Design

The JNI allows for a CryptoNight 'wrapper' class to be created which can provide an interface between external Java applications (such as the Java blockchain server application in this case), and the C++ CryptoNight code. Figure 3.7 shows a diagram outlining the overall design of the CryptoNightJNI wrapper in regards to the JVM, operating system and C++ CryptoNight binaries.

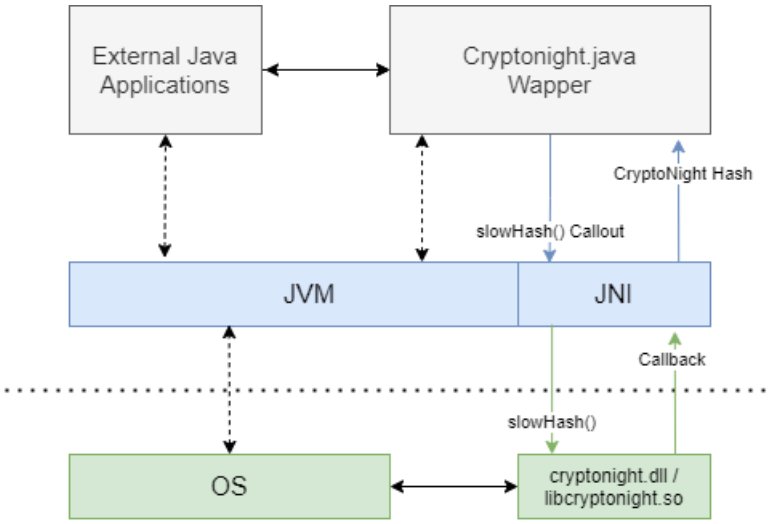


Figure 3.7: CryptoNight Java/C++ JNI wrapper overview

The topmost level in grey shows the interaction between the Cryptonight.java class and other external Java applications. The same constructor initialisation and return methods as the native Java implementation (see section 3.2.2.1) is to be used as it allows for the two components to be easily interchangeable.

Both the external Java applications and the CryptoNightJNI wrapper interact with the operating system through the JVM, with the CryptoNightJNI wrapper utilising the JNI API as well. A callout to the method `slowHash()` (the C++ method name that carries out the CryptoNight hashing) is passed from the Java wrapper class to the JNI, which subsequently makes method call within the C++ binary. There are two binaries provided for Win.x64 and Unix.x64 architectures, for which the wrapper class selects the correct binary to use based on the operating system on initialisation. The binary is then executed directly through the

operating system with the input data provided by the JNI, which subsiquently returns the generated hash as a callback to the JNI (shown in green in Figure 3.7). The JNI interpreters the data and returns it back to the CryptoNight wrapper in Java datatype format. The external Java applications can then call the return method on the wrapper to get the hash.

3.2.3.2 CryptoNightJNI Implementation

The first part of the CryptoNightJNI implementation involves the `Cryptonight.java` wrapper class, which can be seen in Listing 3.14.

```
1  public class Cryptonight {
2      private byte[] out = new byte[32];
3
4      public Cryptonight(String inputData) {
5          Hasher.slowHash(inputData.getBytes(), out);
6      }
7
8      public byte[] returnHash(){
9          return out;
10     }
11 }
```

Listing 3.14: Cryptonight.java unit tests

As discussed previously, the hash is calculated within the constructor, and is returned through the `returnHash()` method, which follows the same structure as the CryptoNightJ solution. Since the output array must be provided prior to calling as the hashing function as is required by the JNI, it is initialised with 32 bytes. The `Hasher.slowHash()` method is the function bound to the external binaries through the JNI which can be seen in Listing 3.15. The `native` keyword intructs the JNI to treat the method as a form of abstract class, however instead of being implemented within Java, it will be implemented within a separate library.

```
1  //C++ library function binding
2  public static native void slowHash(byte[] input, byte[] output);
3
4  static {
5      String binary = "";
6      //Get the system OS
7      final String system = System.getProperty("os.name").toLowerCase();
8
9      try {
10         //Load the binary corresponding with the system architecture
11         if (system.contains("win")) {
12             loadBinary("/win/cryptonight.dll");
13         } else if (system.contains("nix")
14                 || system.contains("nux")
15                 || system.contains("aix")) {
16             loadBinary("/unix/libcryptonight.so");
17         } else {
18             //If the operating system isn't supported, throw exception
19             throw new hasher.UnsupportedOperatingSystemException(String.
       format("Operating system: '%s' is not supported", system));
20         }
21     } catch (Exception e) {
22         e.printStackTrace();
23     }
24 }
```

Listing 3.15: `Hasher.slowHash()` Java binding and library selection

The libraries and bindings are modified from netindev's Monero miner, which works on the CryptoNight V2.2 implementation from the Monero source code. The `Hasher.java` class shown in Listing 3.15 first determines the operating system the application is currently running on before loading its corresponding binary (lines 7 to 16). If the operating system isn't supported (only Windows and Linux binaries are available) an exception is thrown (line 19). The `loadBinary()` method utilises the `class.getResourceAsStream()` method to load the respective binary into the class to be used by the JNI. Listing 3.16 shows the JNI bindings within the C++ code that is compiled to produce the two binaries.

```
1  JNIEXPORT void JNICALL Java_hasher_Hasher_slowHash(JNIEnv *env, jclass
       clazz, jbyteArray input, jbyteArray output) {
2
3      unsigned char* inputBuffer = as_unsigned_char_array(env, input);
4      unsigned char* outputBuffer = as_unsigned_char_array(env, output);
5
6      cryptonight_ctx* ctx[SIZE];
7      for (int i = 0; i < SIZE; i++) {
8        ctx[i] = alloc_ctx();
9      }
10
11     Cryptonight_hash<FUNC>::template hash<cryptonight_monero_v8, false,
       false>(inputBuffer, env->GetArrayLength(input), outputBuffer, ctx);
12
13     for (int i = 0; i < SIZE; i++) {
14       cryptonight_free_ctx(ctx[i]);
15     }
16
17     env->ReleaseByteArrayElements(output, (jbyte *)outputBuffer,
       JNI_COMMIT);
18 }
```

Listing 3.16: C++ JNI binding compiled within the binaries (netindev, 2018)

The `JNIEXPORT` keyword within Listing 3.16 (line 1) is what marks the function as being exportable and thus available to the JNI. The `JNICALL` keyword specifies the method (with its respective class package) which is being linked from the Java class through the JNI. The arguments are defined as `jbyteArray` which dentotes Java byte array data types.

The input and output arguments are converted into C++ chars as they are the equivalent byte data type which is used within the CryptoNight hashing function (lines 3 to 4). The CryptoNight hashing function is then executed with the return being stored within the `outputBuffer` variable (line 11).

The `ReleaseByteArrayElements()` method within the `JNI_COMMIT` keyword instructs the JNI to release the return `outputbuffer` (contains the CryptoNight hash) back to the original output argument which is converts it back into Java byte array format (line 17).

3.2.3.3 Publishing Artifacts to Maven Central Repository

Since this is the first optimal and modular Java wrapper solution for the CryptoNight algorithm, it is important that the artifacts are easily available for other developers to utilise. This one of the main principles behind open source development as its highly inefficient for every developer in the community to recreate a component that was already implemented by someone else, the phrase 'don't reinvent the wheel' often being used in this context. Therefore, the artifacts produced have been uploaded to the Maven central repository and can be easily imported into any projects POM file using the dependency element shown in Listing 3.17. The dependency is also required for this projects CI pipelines as they are running on external cloud platforms (see section 3.3.7).

```
1 <dependency>
2     <groupId>com.jounaidr</groupId>
3     <artifactId>CryptoNightJNI</artifactId>
4     <version>1.1</version>
5 </dependency>
```

Listing 3.17: CryptoNightJNI Maven POM dependency

The process of uploading the artifacts first involved requesting and reserving a group in Maven's sonatype nexus repository which required verification of the domain name used. The artifacts also needed to be signed through GPG, for which the public key needed to be distributed to multiple key servers. After receiving confirmation from a sonartype Jira ticket the release could finally be promoted through the Maven deploy goal. Subsequent releases have their version/subversion incremented, with the release assets being available on the GitHub repository as well the version history being visible through Maven's repository search.

3.3 Blockchain P2P Network

The following section documents the design, implementation and unit testing of the P2P blockchain network server application, that implements the Java CryptoNight solutions created. There are three components that make up the project: the core blockchain backed (section 3.3.2); the server RESTful API (section 3.3.3) and the peer communication and interaction (section 3.3.4). The repository containing all the code discussed, as well as a README file explaining how to install and run the application can be found at: `https://github.com/jounaidr/jrc-node`.

3.3.1 Server Application Design and Overview

Figure 3.8 shows how each of the three application components: the core blockchain backend, the node API, and the peer communication each interact within a node of the network. The dashed arrows represent external interactions with other nodes and applications utilising the API, with the solid arrows denoting internal server component interactions.

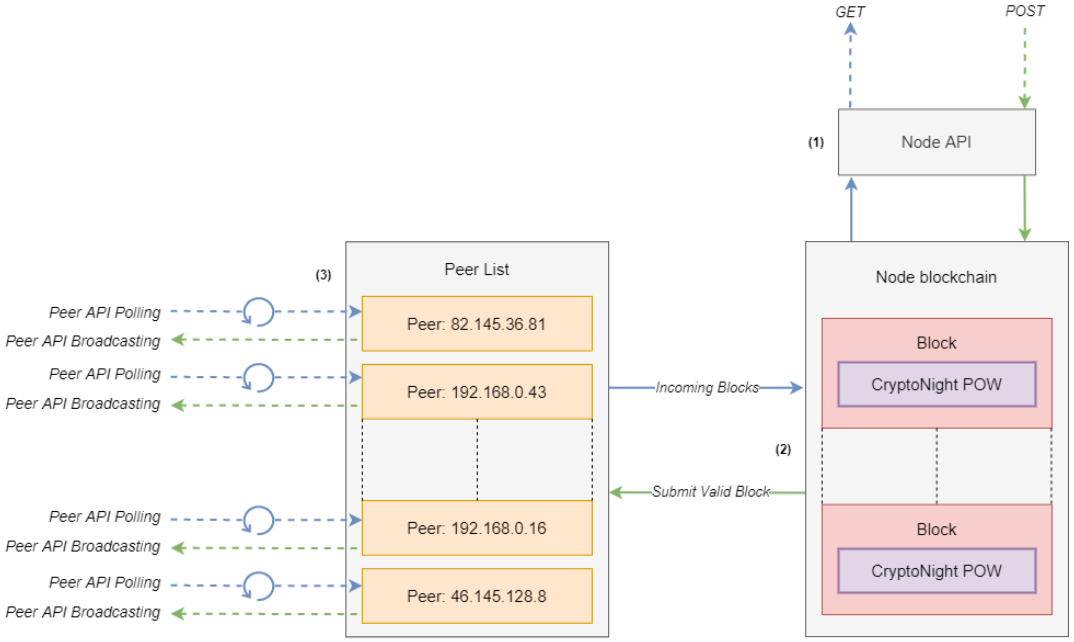


Figure 3.8: Initial server application and networking design

Figure 3.8 section (1) shows the node API that handles all inwards (`POST`) and outwards (`GET`) requests to the nodes blockchain. It is designed with multiple `GET` requests for retrieval of: the nodes entire blockchain; the nodes blockchain length; the nodes last block; the nodes peer list, and a general server 'health' status. The API is also designed with the ability to `POST` potential blocks to the nodes blockchain, for which the node will validate before adding it to its own local chain.

The core blockchain structure shown in Figure 3.8 section (2) is designed with a blockchain object which contains a list of block objects, with methods to add, retrieve and replace said blocks and blockchain (which are used by the API). The block class is designed to have 'factory' constructor methods for block instantiation which create and return block objects within the constructor itself. The first of which is to a return a block with the genesis data, and the second is to generate a next block given a supplied previous block, which requires the

generation of the proof of work. This is where the CryptoNight component is implemented, for which either CryptoNightJ or CryptoNightJNI could be used as they are interchangeable. The class is also designed to have various methods to help with validation around the hash and proof of work, as well as various getter and setter methods for all the blocks properties.

Figure 3.8 section (3) shows the peer communication component of the server application. It is designed to have a peers object that contains a list of peer objects, each of which contains a reference to a peer's socket, as well as client methods to make API requests to said peer. Each peer object is designed to have two 'peer services', a broadcasting service and a polling service.

The polling service is responsible for making `GET` requests to the peer and will continuously 'poll' said endpoints looking for changes in the peer's blockchain and peer list. If new blocks are discovered in a peer, they will be submitted to the nodes local blockchain to be validated (utilising the same methods as the API `POST` block request), and if valid said block will be appended to the local blockchain. The polling service also monitors and keeps track of what peers the peer knows. If new peers are discovered and are healthy, a new peer object will be instantiated for said peer, and will be added to the nodes local peer list and peers object. This is referred to as 'peer discovery' and heavily contributes to the interconnectedness of the network, ensuring all nodes have a 'path' to each other, which is one of the main problems distributed P2P networks face.

When the node's local blockchain has a valid block appended to it either through an API request or from a peer, it is broadcast to each peer in the nodes peers object. This is done by making a `POST` request on said peer's APIs, for which the peer will validate the block itself before adding it to its local blockchain. Through these mechanisms blocks can quickly propagate through the network which is crucial as the whole network needs consensus on the blockchain before the next block is produced.

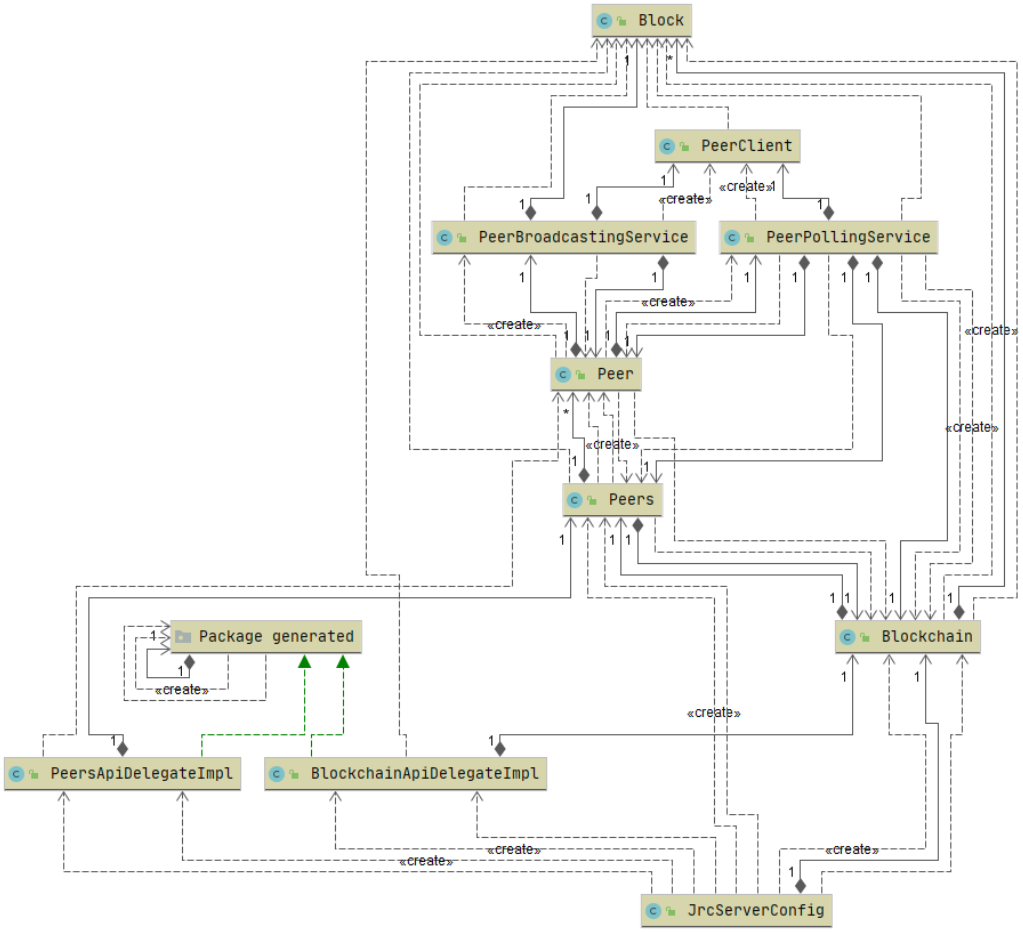


Figure 3.9: Overview of server application implementation

Figure 3.9 shows an overview class diagram of how the different server application components are implemented. At the bottom of the diagram, the `JrcServerConfig` class is essentially what controls everything, from initialisation of all the different beans to injecting properties from external configuration classes. This class is what Spring uses when initialising the server on startup (see section 3.3.5). The two `ApiDelegateImpl` classes to the left of the diagram are beans which handle the API requests that implement the interface classes generated by the OpenAPI generator (2015) (see section 3.3.3). The `Blockchain` bean on the right is what holds the nodes local blockchain list, and the methods to add and retrieve blocks from said local blockchain (see section 3.3.2). The last bean shown in the center of the diagram is `Peers`, which handles the nodes local peer list and `Peer` objects. As explained in the server design, each peer object has a `PeerBroadcastingService` and `PeerPollingService` which interface with the peer using the `PeerClient`. The `Block` class shown at the top of the diagram is rather multipurpose and therefore is used throughout the different classes and beans within the application. A full UML class representation of Figure 3.9 detailing the class properties and methods (including utility/helper classes) can be seen in Appendix D. Full JavaDoc documentation that details the specifics of each class and method can also be viewed at: `https://jounaidr.github.io/jrc-node-javadocs/`.

3.3.2 Core Blockchain Structure

The blockchain package is essentially the core of the application, with the other components being built around the classes within it. Figure 3.10 shows a dependency diagram of specifically the classes that make up the blockchain package, as well as their respective unit test classes.

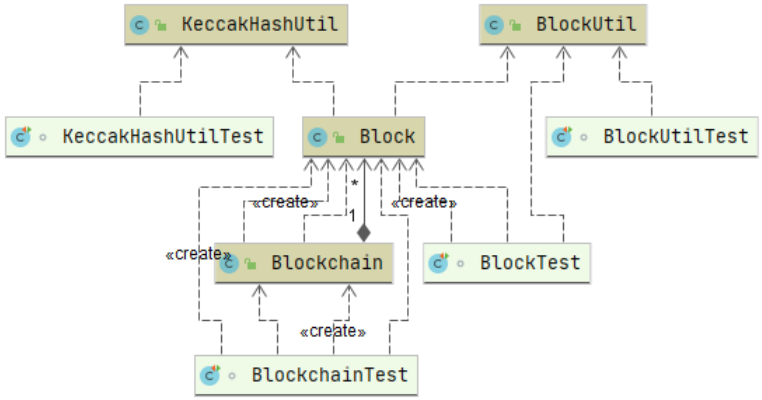


Figure 3.10: Blockchain package class diagram

3.3.2.1 Block Class

As mentioned in the design section, a lot of the applications functionality is in the `Block` class itself. Listing 3.18 shows how the block objects are instantiated from the class, with args, noargs and the genesis factory constructors.

The blocks properties can be seen in lines 10 to 16 and include: `hash`, `previousHash`, `data`, `timeStamp`, `nonce`, `difficulty` and `proofOfWork`. Note that all the properties are stored as string data types and are converted to different datatypes on use.

If the noargs constructor is called, these parameters are set to null which is used during testing (lines 18 to 26). A block can also be instantiated with arguments provided for each of the listed properties using the args constructor (lines 28 to 36). The args constructor is used when converting blocks between formats such as JSON further on in the application. Lines 44 to 55 shows the `genesis()` factory method which returns a block using the hard coded genesis properties (lines 1 to 8).

```
private static final long MINE_RATE = 120;

private static final String GENESIS_PREVIOUS_HASH = "dummyhash";
private static final String GENESIS_DATA = "dummydata";
private static final String GENESIS_TIME_STAMP = "2020-11-07T19
    :40:57.5855811002";
private static final String GENESIS_NONCE = "dummydata";
private static final String GENESIS_DIFFICULTY = "3";
private static final String GENESIS_PROOF_OF_WORK =      "
    1101011101110100010010011001011010101001000010001011 1...";

private String hash;
private String previousHash;
private String data;
private String timeStamp;
private String nonce;
private String difficulty;
private String proofOfWork;

public Block() {
    this.hash = null;
    this.previousHash = null;
    this.data = null;
    this.timeStamp = null;
    this.nonce = null;
    this.difficulty = null;
    this.proofOfWork = null;
}

public Block(String hash, String previousHash, String data, String
    timeStamp, String nonce, String difficulty, String proofOfWork) {
    this.hash = hash;
    this.previousHash = previousHash;
    this.data = data;
    this.timeStamp = timeStamp;
    this.nonce = nonce;
    this.difficulty = difficulty;
    this.proofOfWork = proofOfWork;
}

/**
 * Use genesis constants to generate the
 * genesis block
 *
 * @return the genesis block
 */
public Block genesis(){
    this.setPreviousHash(GENESIS_PREVIOUS_HASH);
    this.setData(GENESIS_DATA);
    this.setTimeStamp(GENESIS_TIME_STAMP);
    this.setDifficulty(GENESIS_DIFFICULTY);
    this.setNonce(GENESIS_NONCE);
    this.setProofOfWork(GENESIS_PROOF_OF_WORK);

    this.setHash(this.generateHash()); //Generate the genesis block hash

    return this;
}
```

Listing 3.18: Block class constructors and genesis factory method

Listing 3.19 shows the `mineBlock()` factory method within the block class. This method is used to generate a new block with some data given a previous block, and as such the proof of work must be generated for the block. This is done by continuously generating CryptoNight hashes with the block properties until a hash that has a number of proceeding 0's equivalent to the difficulty level is generated (lines 8 to 24).

In order for a different hash to be generated in each iteration of the loop, a nonce value (`currentNonce` on line 9) is incremented. This is done to prevent 'duplication of work' which would occur if the same hash were to be difficulty checked twice.

Once the loop has concluded and a valid proof of work string has been found, the `nonce` and `proofOfWork` properties for the block are set, and with all properties available the blocks `hash` is generated (lines 26 to 68). A final validity check on the block is carried out (lines 30 to 35) before the freshly mined block is returned.

```
1  public Block mineBlock(Block previousBlock, String data){
2      this.setPreviousHash(previousBlock.getHash());
3      this.setData(data);
4
5      int currentNonce = 0;
6      byte[] currentProofOfWork;
7
8      do{
9          currentNonce++;
10
11         Instant ts = Instant.now();
12         //Set timestamp to current time and current interation in loop
13         this.setTimeStamp(ts.toString());
14
15         //Adjust the difficulty based on the new timestamp
16         adjustDifficulty(previousBlock);
17
18         String proofOfWorkData = this.previousHash + this.data + this.
       timeStamp + this.difficulty + currentNonce;
19
20         Cryptonight cryptonightPOW = new Cryptonight(proofOfWorkData);
21         currentProofOfWork = cryptonightPOW.returnHash();
22
23     //Check if the currently calculated proof of work leading zeros meets
       the difficulty
24     } while(!(this.difficulty.equals(String.valueOf(BlockUtil.
       getByteArrayLeadingZeros(currentProofOfWork)))));
25
26     this.setNonce(String.valueOf(currentNonce));
27     this.setProofOfWork(BlockUtil.getBinaryString(currentProofOfWork));
28     this.setHash(this.generateHash());
29
30     try {
31         // Validate the newly mined block
32         this.validateBlock(previousBlock);
33     } catch (InvalidObjectException e) {
34         e.printStackTrace();
35     }
36
37     return this;
38 }
```

Listing 3.19: Block.java `mineBlock()` method

Within the `mineBlock()` method difficulty adjustment must take place in order to accommodate for the total mining power of the network. This is because as the more miners are generating blocks in the network, the chance a valid proof of work is found increases, resulting in blocks being added more frequently in the network. In order to prevent this, a target block time of 120 seconds is defined in the `MINE_RATE` constant (see Listing 3.18 line 1), which the `difficulty` property is adjusted based on, with the goal of producing blocks in said block time.

The `adjustDifficulty()` method that carries out this mechanism can be seen in Listing 3.20. The `BlockUtil.calcBlockTimeDiff()` method is used to calculate the difference between the current block and the previous block, which does a datetime conversion in order to calculate the time difference in seconds (line 15). If the calculated time difference is greater than the `MINE_RATE`, the difficulty is decremented to make the next block's proof of work easier to find (lines 17 to 20). If the time difference is less than the target `MINE_RATE`, the difficulty is incremented which has the opposite effect, increasing the time taken to generate the next blocks proof of work (lines 21 to 24).

There are final validation check to ensure the difficulty isn't negative (lines 26 to 29) before the adjusted difficulty is set on line 31.

```
1  /**
2   * Method calculates the time diff between the
3   * previous block and block currently being mined
4   * and will increment the difficulty if diff is
5   * less than the MINE_RATE, and decrement difficulty
6   * if diff is greater than the MINE_RATE
7   * Method also checks for negative difficulty and
8   * will set difficulty to 1 if this should happen
9   *
10  * @param previousBlock the previous block
11  */
12 private void adjustDifficulty(Block previousBlock){
13     int difficulty = Integer.parseInt(previousBlock.getDifficulty());
14     //Difference in seconds between the block currently being mined, and
       the previously mined block
15     long diffSeconds = BlockUtil.calcBlockTimeDiff(this.timeStamp,
       previousBlock.getTimeStamp());
16
17     if(diffSeconds > MINE_RATE){
18         //Decrement difficulty if time taken is greater than MINE_RATE
19         difficulty--;
20     }
21     else{
22         //Otherwise increase difficulty in an attempt to increase block
       mine time
23         difficulty++;
24     }
25
26     if(difficulty < 1){
27         log.error("Previous block difficulty is negative: {}, setting
       current block difficulty to 1...", difficulty);
28         difficulty = 1; //Set difficulty to 1 if it drops below 1
29     }
30
31     this.setDifficulty(String.valueOf(difficulty));
32 }
```

Listing 3.20: Block.java `adjustDifficulty()` method

Its important to note that the `proofOfWork` and `hash` properties are separate despite both essentially representing the same thing. The reason why this is done is because CryptoNight is a relatively slow hashing method, so when the blockchain becomes increasingly large it will take a large amount of time to validate each block in the chain using this hashing algorithm. Therefore the blocks hash that is referenced in the next block is generated using SHA-3 (keccak) which is much faster than the CryptoNight algorithm.

The `KeccakHashUtil.returnHash()` utility method is used to generate the SHA-3 hash which implements the bouncy castle (The Legion of the Bouncy Castle, 2000) provided hashing methods. Said utility method in the `KeccakHashUtil` class can be seen in Listing 3.21.

```
/**
 * Implementation of Keccak-256 hashing algorithm
 * provided by the Bouncy Castle Library, based on
 * https://www.baeldung.com/sha-256-hashing-java
 * section 6.3
 *
 * @return the hashed message string in lowercase hex format
 */
public static String returnHash(String message){
    Keccak.Digest256 digest256 = new Keccak.Digest256();
    byte[] hashbytes = digest256.digest(message.getBytes(StandardCharsets.
    UTF_8));

    return new String(Hex.encode(hashbytes));
}
```

Listing 3.21: KeccakHashUtil.java SHA-3 wrapper method

The block class also contains two validation methods, `isProofOfWorkValid()` and `validateBlock()`. The `isProofOfWorkValid()` method shown in Listing 3.22 generates a new CryptoNight hash based on the blocks properties and checks whether the hash produced satisfies the difficulty utilising the `BlockUtil.getBinaryString()` (line 12) method in the same way as during proof of work generation.

```
/**
 * Method will regenerate the POW binary string
 * and validate it against the POW string that is
 * set for this block
 *
 * @return if POW is valid
 */
public Boolean isProofOfWorkValid(){
    String proofOfWorkData = this.previousHash + this.data + this.
    timeStamp + this.difficulty + this.nonce;
    Cryptonight cryptonightValidator = new Cryptonight(proofOfWorkData);

    String proofOfWorkBinaryString = BlockUtil.getBinaryString(
    cryptonightValidator.returnHash());

    return this.proofOfWork.equals(proofOfWorkBinaryString);
}
```

Listing 3.22: Block.java `isProofOfWorkValid()` validation method

The `validateBlock()` method shown in Listing 3.23 does five validation checks on the block using a supplied previous block, with an `InvalidObjectException` being thrown on failure of any of said validation checks.

The first two checks simply validate the supplied previous block's hash and proof of work (lines 12 to 18). The third check verifies that the current blocks `previousHash` property correctly references the supplied previous block's hash (lines 21 to 23). The final two checks (lines 25 to 31) validate the current blocks hash through regeneration (see Listing 3.21), and proof of work (see Listing 3.22).

```
1  /**
2   * Method to validate a block. First checks will verify the
3   * provided previousBlock has a valid hash and proof of work,
4   * then will check that the block being validated has a
5   * valid hash, valid proof of work, and that its previous hash
6   * references the provided previousBlocks hash correctly
7   *
8   * @param previousBlock the previous block
9   */
10 public void validateBlock(Block previousBlock) throws
       InvalidObjectException {
11     // Validation checks against the supplied previous block
12     if(!previousBlock.getHash().equals(previousBlock.generateHash())){
13         throw new InvalidObjectException(String.format("Block validation
       failed, supplied previous block has an invalid hash. Supplied previous
       block hash: %s, should be: %s...", previousBlock.getHash(),
       previousBlock.generateHash()));
14     }
15
16     if(!previousBlock.isProofOfWorkValid()){
17         throw new InvalidObjectException("Block validation failed,
       supplied previous block has an invalid proof of work...");
18     }
19
20     // Validation checks for this block
21     if(!this.getPreviousHash().equals(previousBlock.getHash())){
22         throw new InvalidObjectException(String.format("Block validation
       failed, this block doesn't reference the previous blocks hash correctly
       . Reference to previous hash: %s, supplied previous blocks hash: %s..."
       , this.getPreviousHash(), previousBlock.getHash()));
23     }
24
25     if(!this.getHash().equals(this.generateHash())){
26         throw new InvalidObjectException(String.format("Block validation
       failed, this block has an incorrect hash value. This blocks hash: %s,
       should be: %s...", this.getHash(), this.generateHash()));
27     }
28
29     if(!this.isProofOfWorkValid()){
30         throw new InvalidObjectException("Block validation failed, this
       block has an incorrect proof of work...");
31     }
32 }
```

Listing 3.23: Block.java `validateBlock()` validation method

3.3.2.2 Blockchain Class

The blockchain class is somewhat simple as it only contains a list of block objects with methods to interact with said list, however since its a core part of the application with multiple components accessing the same list, thread safety measures are implemented to ensure there are no concurrency issues. Listing 3.24 shows the blockchain class constructor method as well as the `chain` method getters/setters.

```
1  @Autowired
2  Peers peers;
3
4  private List<Block> chain;
5
6  private ReadWriteLock rwLock = new ReentrantReadWriteLock();
7  private Lock readLock = rwLock.readLock();
8  private Lock writeLock = rwLock.writeLock();
9
10 public Blockchain(List<Block> chain) {
11     log.debug("Attempting to initialise a blockchain with the following
       chain array: {}...", chain.toString());
12     this.setChain(chain);
13
14     if(this.getChain().size() < 1){
15         this.getChain().add(new Block().genesis());
16         log.info("A Fresh blockchain has been initialised with genesis
       block...");
17         log.debug("Blockchain initialised with the following genesis block
       : {} ...", this.getLastBlock().toString());
18     }
19 }
20
21 public List<Block> getChain() {
22     //Read lock whilst getting chain as many threads can read/write to
       this
23     readLock.lock();
24
25     try {
26         return this.chain;
27     } finally {
28         readLock.unlock();
29     }
30 }
31
32 private void setChain(List<Block> newChain) {
33     //Write lock whilst setting chain as many threads can read/write to
       this
34     writeLock.lock();
35
36     try {
37         this.chain = newChain;
38     } finally {
39         writeLock.unlock();
40     }
41 }
```

Listing 3.24: Blockchain.java constructor and `chain` getters/setters

There are five variables initialised within the blockchain class: `peers`, `chain`, `rwLock`, `readLock` and `writeLock` (lines 1 to 8). The `peers` object is declared with an `@Autowired` tag that

instructs spring to inject the peers service bean into the local variable.

As this class is ran a service bean, the constructor will only be ran once as there will only ever be one blockchain object instantiated within the application. A constructor argument containing the chain array is required, which in production is supplied an empty ArrayList (see section 3.3.5) that will subsequently have a genesis block added to it (line 15). The reason this class is not simply initialised with an empty ArrayList within the constructor is to increase testability, as invalid chains can be injected into the class easily.

When a thread attempts to read the chain array through the getChain() method, the thread will 'acquire' a readLock (line 23), which is subsequently released once the read has finished (line 28). When a thread attempts to write to the chain array through the setChain() method, the thread will acquire a writeLock. The writeLock can only be acquired if there are no other read or write locks, and only one writeLock can be acquired at a time (unlike the readLock where multiple threads can acquire said lock concurrently). The writeLock is released once the thread has finished modifying the chain array (line 30), which allows for other threads to begin acquiring locks again. This prevents any data inconsistency issues that could occur if multiple threads are trying to access/modify the array at the same time. Due to the ability for multiple threads to read a shared resource at the same time with ReentrantReadWriteLock(), it is much more performant than other thread safety mechanisms such as Synchronised methods or stamped locks, which completely lockdown the shared resource on either reads or writes.

```
1  /**
2   * Validate an new incoming block and if valid,
3   * add the block to the chain
4   *
5   * @param newBlock the new incoming block
6   */
7  public void addBlock(Block newBlock) throws InvalidObjectException {
8      log.info("A new block has been submitted to the blockchain!");
9      // Check if the block is has already been added
10     if(newBlock.toString().equals(this.getLastBlock().toString())){
11         log.info("Block has already been added to the chain!");
12         return;
13     }
14
15     log.info("Attempting to add new incoming block: {}...", newBlock.
       toString());
16     try {
17         // Validate the incoming block against this blockchains last block
        before adding new block
18         newBlock.validateBlock(this.getLastBlock());
19         this.getChain().add(newBlock);
20         log.info("...Block added successfully!");
21         // Then broadcast the new block to the nodes peers
22         peers.broadcastBlockToPeers(newBlock);
23     } catch (InvalidObjectException e) {
24         log.error("New incoming block is invalid and can't be added to the
        blockchain. Reason: {}", e.getMessage());
25         throw e;
26     }
27 }
```

Listing 3.25: Blockchain.java addBlock method

Listing 3.25 shows the method used to add new blocks to the chain array. An initial check to confirm that the block has not allready been added to the chain is first carried out (lines

10 to 12). If the block is new to the local blockchain, it will be first validated using the `validateBlock()` method (see Listing 3.23) with the previous last block in the `chain` array (line 18), before being appended to said array. The new block is then broadcast to all peers known to the node using the `broadcastBlockToPeers()` (see Listing 3.33) method within the `peers` service bean (line 22).

```
1  public void replaceChain(Blockchain newBlockchain) throws
       InvalidObjectException {
2      log.info("Attempting to replace the current blockchain with a new
       incoming blockchain");
3
4      if(!newBlockchain.isChainValid()){
5          log.debug("Incoming blockchain is longer than current blockchain,
       but is not valid...");
6          return;
7      }
8
9      this.setChain(newBlockchain.getChain());
10     log.info("Chain replacement successful...");
11 }
12
13 public boolean isChainValid() throws InvalidObjectException {
14     if(!(this.getChain().get(0).toString()).equals(new Block().genesis().
       toString())){
15         log.error("Chain is invalid, first block in the chain is not
       genesis block...");
16         return false; //Verify first block in chain is genesis block
17     }
18
19      for(int i=1; i < this.getChain().size(); i++){
20          try {
21            //Verify each block is valid against the previous block
22            this.getChain().get(i).validateBlock(this.getChain().get(i-1));
23          } catch (InvalidObjectException e) {
24            log.error("Chain is invalid, the block {} in the chain is
       invalid.",i);
25            throw e;
26          }
27
28         //Verify each block changes the difficulty by no more than 1
29         int changeInDifficulty = Math.abs(Integer.parseInt(this.getChain()
       .get(i-1).getDifficulty()) - Integer.parseInt(this.getChain().get(i).
       getDifficulty()));
30
31         if(changeInDifficulty > 1){
32             log.error("Chain is invalid, the block {} in the chain has a
       difficulty jump greater than 1. Difficulty changed by: {}...",i,
       changeInDifficulty);
33             return false;
34         }
35     }
36     log.debug("Blockchain is valid...");
37     return true;
38 }
```

Listing 3.26: Blockchain.java whole chain validation and replacment methods

Listing 3.26 shows the whole blockchain validation and replacement methods. In the occasion where the nodes blockchain becomes out of sync with the network and is multiple blocks

behind, the whole `chain` array must be replaced. This is done in the `replaceChain()` method (lines 1 to 11) which first validates the chain before calling the `setChain()` method to replace the `chain` array.

The blockchain validation takes place in the `isChainValid()` method (line 13). The first check is to confirm that the blockchain has the correct genensis block (lines 14 to 17). If successfully, every block in the blockchain will be consecutively validated using the `validateBlock()` method (see Listing 3.23) with the previous block in the blockchain supplied (lines 19 to 26). The difficulty of each block in the blockchain is also checked to ensure it doesn't increase by more than 1 between blocks, known as a 'difficulty jump' (lines 29 to 34). If all validation checks pass successfully, the method will return true denoting a valid blockchain.

3.3.3 Server API

The Spring Web dependency initialises a Tomcat container within the application, which allows provides a HTTP web server environment for the Java code to run within. This is combined with the OpenAPI generator, which simplifies the implementation of API methods using a consolidated specification file in either YAML or JSON format (YAML chosen within this application). This specification file can also be used to generate client and server code in various different languages allowing for easy creation of third party components, a full list of which can be found in the OpenAPI generator documentation. Said specification file can also be used to generate a HTML documentation of the API in an interactive and readable format, for which the generated documentation for this API can be found at:
`https://jounaidr.github.io/jrc-node-API-docs/`

The full API specification file can be seen in Appendix E. The first section (lines 3 to 15) contains general information about the API such as a title and description as well as licensing and version information. It also contains a link to the project repository. The version property is particularly important as it ensures two applications are communicating using the same endpoint formats (if said endpoints change between versions).

The next section contains the endpoint definitions (lines 20 to 97) which is whats used by the OpenAPI generator to create the implementation classes. Each endpoint is defined with a short descriptive summary, an `operationId` (which becomes the endpoints implementation method name) and possible responses. Each response is defined with a response code that conforms to the Internet Assigned Numbers Authority (IANA), HTTP Status Code Registry (Fielding and Reschke, 2014), for which codes 200 (OK) and 400 (Bad Request) are used.

The final section contains model definitions which are used by the OpenAPI generator to create model classes. These are used when converting the raw JSON requests/responses to workable objects within Java (or whatever language client/server code is generated).

There are four endpoints which provide an interface with the nodes blockchain. The top level endpoint `GET /blockchain` returns the entire blockchain in the format defined by the `Blockchain` schema (line 104 to 107) which is just an array of `BlockModel` objects. The `GET /blockchain/lastblock` endpoint returns the last block in the chain, for which the response format is defined in the `BlockModel` schema (lines 109 to 125) which is just an object containing all the block properties as strings. The `POST /blockchain/addblock` endpoint takes a request body in JSON format as defined by the `BlockModel` schema, an example of which can be seen in Listing 3.27. If the block is successfully added, a 200 response is sent with a successes message, and if the block is unsuccessfully added due to the block being invalid, a 400 response is sent with the exception message detailing why the block was rejected. There is one endpoint related to the known peers of the node, `GET /peers`, which returns a list of `PeerModel` objects that just contain each peer's socket and health status (lines 127 to 138).

```
{
    "hash": "ba2dc038b230484912964f151cf53de469c3271ddafdd91963cfa8974ea8c11b",
    "previousHash": "89b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22",
    "data": "test",
    "timeStamp": "2020-12-29T19:56:44.524696500Z",
    "nonce": "10",
    "difficulty": "2",
    "proofOfWork": "00110111100010001010101101100111110001001001011001000000001..."
}
```

Listing 3.27: Example JSON block payload

The specification file is placed in a separate module called `server-api-generator`, for which the generated files are built through maven and placed in the `com.jounaidr.jrc.server.api.generated` package. The OpenAPI maven plugin implementation within the POM file can be seen in Listing 3.28.

```
<groupId>org.openapitools</groupId>
<artifactId>openapi-generator-maven-plugin</artifactId>
<version>4.3.1</version>
<execution>
    <goals>
        <goal>generate</goal>
    </goals>
    <configuration>
        <inputSpec>
            ${project.basedir}/src/main/resources/openapi.yaml
        </inputSpec>
        <generatorName>spring</generatorName>
        <apiPackage>
            com.jounaidr.jrc.server.api.generated
        </apiPackage>
        <modelPackage>
            com.jounaidr.jrc.server.api.generated.model
        </modelPackage>
        <supportingFilesToGenerate>
            ApiUtil.java
        </supportingFilesToGenerate>
        <configOptions>
            <delegatePattern>true</delegatePattern>
        </configOptions>
    </configuration>
</execution>
```

Listing 3.28: OpenApi generator maven plugin

The dependency structure the generated classes (and their implementations) follow can be seen Figure 3.11, with the green arrows denoting interface implementation. The API controller classes are what interact with the Tomcat container through Spring to process the server requests/responses. An overview of the dependencies between the generated classes package as a whole and the rest of the server application classes can be seen in Figure 3.9.

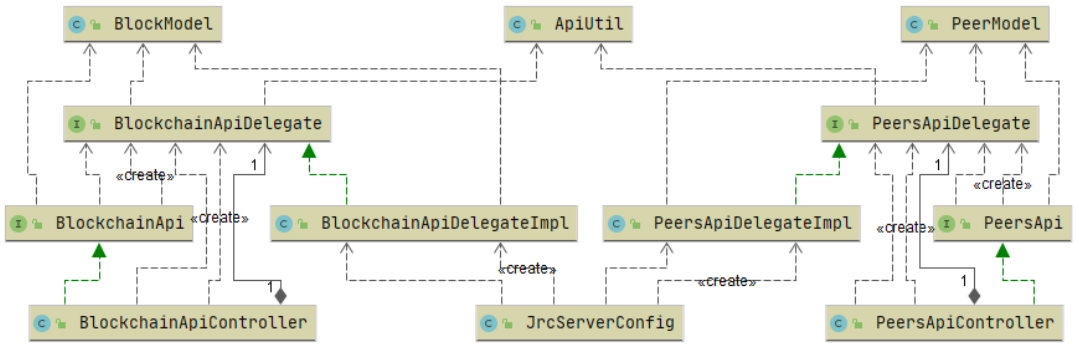


Figure 3.11: Dependency diagram of generated API classes and implementations

Of the generated classes, API delegate interfaces are created for both the blockchain and peers endpoints which are named `BlockchainApiDelegate` and `PeersApiDelegate` re-

spectively. These interfaces contain a method for each of the defined endpoints in the specification file, which as mentioned are named by their `operationId`. The interfaces are then implemented, with each method overwritten with its corresponding response either from the blockchain or peers service bean. Said implementation classes are named `BlockchainApiDelegateImpl` and `PeersApiDelegateImpl` and are also run as service beans. The methods within `BlockchainApiDelegateImpl` can be seen in Listing 3.29.

```
1  @Autowired
2  private Blockchain blockchain;
3
4  @Override
5  public ResponseEntity<List<BlockModel>> getBlockchain() {
6      ArrayList<BlockModel> response = new ArrayList<>();
7
8      for(Block block : blockchain.getChain()){
9          response.add(BlockModelUtil.getBlockAsModel(block));
10     }
11
12     return ResponseEntity.ok(response);
13 }
14
15 @Override
16 public ResponseEntity<Integer> getBlockchainSize() {
17     Integer response = blockchain.getChain().size();
18
19     return ResponseEntity.ok(response);
20 }
21
22 @Override
23 public ResponseEntity<BlockModel> getLastBlock() {
24     BlockModel response = BlockModelUtil.getBlockAsModel(blockchain.
       getLastBlock());
25
26     return ResponseEntity.ok(response);
27 }
28
29 @Override
30 public ResponseEntity<Object> addBlock(BlockModel newBlock){
31     try {
32       blockchain.addBlock(BlockModelUtil.getBlockFromModel(newBlock));
33     } catch (InvalidObjectException e) {
34       return new ResponseEntity<>(e.getMessage(), HttpStatus.BAD_REQUEST);
35     }
36
37     return ResponseEntity.ok("Block added successfully!");
38 }
```

Listing 3.29: Implementation of generated `BlockchainApiDelegate` interface class

The blockchain service bean is injected into the class through Spring using an `@Autowired` tag (line 2) which is used to access the local blockchain throughout the implemented methods. The `getBlockchain()` method converts each block object in the blockchain's `chain` array into a `blockModel` utilising the `getBlockAsModel()` method (lines 8 to 10). This list of block models is then returned as a response entity with a 200 response code. The `getBlockchainSize()` method simply returns the size of the `chain` array as a response entity with a 200 response code (lines 15 to 20). Similar to the `getBlockchain()` method, the `getLastBlock()` method gets the last block in the blockchains `chain` array and coverts

it into a blockModel object before returning it as a response entitiy with response code 200 (lines 22 to 27). The final implemented method is the addBlock() method. Since its a POST request, a response body is received and needs to be handled, which is done through the constructor as a blockModel object (line 30). The model is converted into a usable block object utilising the BlockModelUtil.getBlockFromModel() method before its added to the blockchain (line 32). If the block is rejected (see Listing 3.25 and Listing 3.23) the corresponding exception message is returned as a response entity with response code 400 (line 34). If the block is added to the blockchain, a success message along with response code 200 is sent (line 37).

```
1  public class PeersApiDelegateImpl implements PeersApiDelegate {
2      @Autowired
3      private Peers peers;
4
5      /**
6       * For each peer object in the peers peerList, convert it into
7       * a PeerModel data object and store in temp arraylist to be returned
8       *
9       * @return the PeerModel list as a response entity with status code
       200
10      */
11     @Override
12     public ResponseEntity<List<PeerModel>> getSocketsList() {
13         ArrayList<PeerModel> response = new ArrayList<>();
14
15         for(Peer peer : peers.getPeerList()) {
16             response.add(PeerModelUtil.getPeerAsModel(peer));
17         }
18
19         return ResponseEntity.ok(response);
20     }
21 }
```

Listing 3.30: Implementation of generated PeersApiDelegate interface class

The PeersApiDelegateImpl only has one method as there is only one endpoint related to peers, which can be seen in Listing 3.30. The peers service bean is injected through spring using an @Autowired tag, which is then used in the getSocketsList() method (lines 12 to 19) to return a list of peerModel objects with response code 200. Each peerModel object simply contains the peers socket and health status. Examples of each methods JSON responses after the response entities have been processed through Spring can be seen in the API documentation.

3.3.4 Peer Communication

A crucial part of the server application are the classes that provide interaction with other nodes APIs, for which the implementation is located within the peers package. An expanded class dependency diagram of said package can be seen in Figure 3.12.

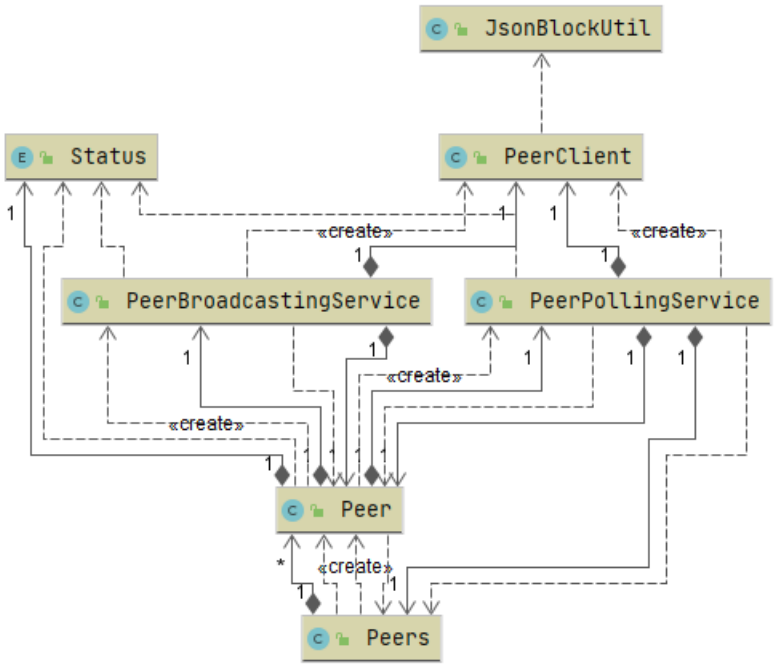


Figure 3.12: Class dependency diagram of peers package

The `Peers` class is run as a service bean and is what maintains the list of peer objects known to the node. This bean is initialised with a list of sockets from the external properties file (see section 3.5.5). The peers service bean is also what maintains the thread pool which has a maximum number of threads equal to the number of peer objects in the peer list. Each peer object has a health status, as well as a `PeerBroadcastingService` and `PeerPollingService` which handle the connectivity of the peer (threaded). The actual requests and subsequent responses to and from the peer are handled by the `PeerClient` class. This structure/hierarchy is chosen as it allows for minimal code duplication and clear separation of logic, whilst also providing an easy way to add additional services by simply utilising the `PeerClient`.

3.3.4.1 Peers Class

The `Peers` class properties and constructor can be seen in Figure 3.31. Although the blockchain service bean isn't used within this class, it is required by the peer services and since those services are not run as beans they cannot have other beans injected into them through Spring. Therefore, the blockchain service bean is injected at the `Peers` class level (lines 1 to 2), and is passed down through constructor arguments to where its required. Similar to the blockchain service bean, many threads will be accessing the peers service bean and therefore a level of thread safety is required, which is implemented using a `ReentrantReadWriteLock` (lines 4 to 6). Two constants are defined for the `NODE_SOCKET` and `MAX_PEERS` (lines 8 to 9) which are passed in through the external configuration file (see section 3.3.5). These denote the socket for the node itself as well as the maximum number of peers the node can work with.

A ScheduledThreadPoolExecutor is defined (line 11) which holds all the threads the peers are able to use. This thread executor is passed down to the peer polling and broadcasting services through constructor arguments in the same way as the blockchain service bean. Its important to note that all polling/broadcasting services use the same thread executor that is defined in this top level class. An integer denoting the maximum number of threads for said executor is also defined (line 12). The final property is simply an array list that holds the list of Peer objects (line 14).

```
1  @Autowired
2  private Blockchain blockchain;
3
4  private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
5  private final Lock readLock = rwLock.readLock();
6  private final Lock writeLock = rwLock.writeLock();
7
8  private final String NODE_SOCKET;
9  private final int MAX_PEERS;
10
11 private final ScheduledThreadPoolExecutor peersExecutor;
12 private int poolSize;
13
14 private final ArrayList<Peer> peerList;
15
16 /**
17  * Instantiates the peers service which contains a list of
18  * all the nodes known peers, and methods to interact with
19  * the peers
20  *
21  * @param nodeSocket  this nodes socket (used for validation checks)
22  * @param maxPeers    the maximum peers for the system
23  * @param socketsList initial list of peer sockets from properties
24  */
25 public Peers(String nodeSocket, int maxPeers, String socketsList) {
26     this.blockchain = blockchain;
27
28     this.NODE_SOCKET = nodeSocket; //This nodes socket
29     this.MAX_PEERS = maxPeers; //Max number of peers from properties
30
31     if(maxPeers > Runtime.getRuntime().availableProcessors()){
32         log.warn("It is recommended to set the max peers to less than {}
    for your system, performance may be impacted...", Runtime.getRuntime().
    availableProcessors());
33     }
34     //Initialise the executor with a pool size of 1
35     peersExecutor = new ScheduledThreadPoolExecutor(poolSize++);
36
37     //Initialise the array of Peer.java objects, and add Peer's to the
    list for each socket provided in properties
38     this.peerList = new ArrayList<>();
39     this.addSocketsList(socketsList);
40 }
```

Listing 3.31: Peers class constructor

Within the constructor the number of logical cores available to the system is checked, and if it exceeds the maximum number of peers a warning message is displayed (lines 31 to 33). This is because each peer increases the thread pool size by one, meaning scheduling would no longer be handled through the executor if the thread pool size exceeds the number of logical

threads physically available in the CPU. After all parameters are initialised, the sockets list provided from the external configuration file is then used to initialise the initial set of peer objects (line 39). The methods that are used to add new peer objects to the `peerList` array can be seen in Listing 3.32.

```
1  public void addSocketsList(String socketsList){
2      log.info("Attempting to add the following sockets [{}] to the peer
       list", socketsList);
3      if(!StringUtils.isEmpty(socketsList)){
4          // Split the socket list and add a peer for each individual socket
5          for(String peerSocket : socketsList.split(",")){
6              this.addPeer(peerSocket);
7          }
8      }
9  }
10
11 private void addPeer(String peerSocket){
12     if(this.getPeerList().size() >= MAX_PEERS){
13         log.error("Unable to add new peer [{}] as max peer size of {} has
       been reached", peerSocket, MAX_PEERS);
14         return;
15     }
16     if(peerSocket.equals(NODE_SOCKET)){
17         log.error("Unable to add new peer [{}] as its socket refers to
       this node!", peerSocket);
18         return;
19     }
20     if(!isSocketValid(peerSocket)){
21         log.error("Unable to add new peer [{}] as its socket is of invalid
        format", peerSocket);
22         return;
23     }
24     if(isPeerKnown(peerSocket)){
25         log.info("Unable to add new peer [{}] as its already known",
       peerSocket);
26         return;
27     }
28
29     //Write lock whilst adding peer as many threads can add peers
30     writeLock.lock();
31     try {
32      this.peersExecutor.setCorePoolSize(poolSize++);
33      this.peerList.add(new Peer(peerSocket, peersExecutor, blockchain,
       this));
34
35      log.debug("Starting polling with new peer [{}] ...", peerSocket);
36      this.getPeerList().get(this.getPeerList().size() - 1).startPolling();
37     } finally {
38      writeLock.unlock();
39     }
40 }
```

Listing 3.32: Peers class constructor

The `addSocketsList()` method takes in a comma separated string of sockets and splits them in order for each socket to be individually validated and initialised as peer objects using the `addPeer()` method (lines 1 to 9). This method is publicly available and is what is used by the polling service during peer discovery. The `addPeer()` method has four validity measures before the socket is initialised as a peer object. First, the current number of peers is checked

too see if it exceeds the MAX_PEERS set (lines 12 to 15). The socket is then checked against the NODE_SOCKET constant so that the node itself isn't initialised as a peer object (lines 16 to 19). A validity check to the socket's format is carried out using the isSocketValid() method (lines 20 to 23), and the current peerList is scanned to check whether the socket already exists as a peer using the isPeerKnown() method (lines 24 to 27). Before the peer object for the socket is initialised and added to the peerList, a write lock is acquired (line 30) as multiple polling service threads have the ability to add peers through this method. The executors thread pool size is incremented (line 32) before the polling service service for the newly initialised peer object is started (line 36). The isSocketValid() method, as well as the other peers methods can be seen in Figure 3.33.

```
1  private boolean isSocketValid(String peerSocket){
2      //Regex patterns for valid ipv4 and ipv6 sockets
3      String ipV4 = "(\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}):(\\d+)";
4      String ipV6 = "\\[([a-zA-Z0-9:]+)\\]:(\\d+)";
5      Pattern validSocket = Pattern.compile(ipV4 + "|" + ipV6);
6
7      return validSocket.matcher(peerSocket).matches();
8  }
9
10 private boolean isPeerKnown(String peerSocket){
11     for(Peer peer : this.getPeerList()){
12         if(peer.getPeerSocket().equals(peerSocket)){
13             return true;
14         }
15     }
16     return false;
17 }
18
19 public void broadcastBlockToPeers(Block block){
20     for(Peer peer : this.getPeerList()){
21         peer.broadcastBlock(block);
22     }
23 }
24
25 public ArrayList<Peer> getPeerList() {
26     //Read lock as multiple threads can read the peeList
27     readLock.lock();
28
29     try {
30         return peerList;
31     } finally {
32         readLock.unlock();
33     }
34 }
```

Listing 3.33: Socket validity and other Peers methods

In order to check the socket string is of correct format, two regex pattern are defined ipV4 and ipV6 (lines 3 to 4). These are complied using the Pattern.compile() method (line 5) before before being matched against the socket string in the methods return (line 7). The isPeerKnown() method simply iterates through the peerList and does a string check against each peer objects socket and the socket provided (lines 10 to 17). The broadcastBlockToPeers() method, which is used by the blockchain service bean when a new block is added locally (see Listing 3.25), simply iterates through the peerList and calls the breadcastBlock() method within each peer object (lines 19 to 23). When a thread

attempts to the access the peerList through the getPeerList() (lines 25 to 34) method, a read lock is acquired as to mitigate inconsistencies within the array as multiple threads can read/write to the variable.

3.3.4.2 Peer Class

The peers class is somewhat simple as its more of a container for the polling/broadcasting services providing publicly available 'controller' methods. The class constructor aswell as said controller methods can be seen in Listing 3.34.

```
1  private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
2  private final Lock readLock = rwLock.readLock();
3  private final Lock writeLock = rwLock.writeLock();
4
5  private final String PEER_SOCKET;
6  private Status peerStatus;
7
8  private final PeerPollingService peerPoller;
9  private final PeerBroadcastingService peerBroadcaster;
10
11 public Peer(String peerSocket, ScheduledThreadPoolExecutor peersExecutor,
       Blockchain blockchain, Peers peers) {
12     this.PEER_SOCKET = peerSocket;
13     this.peerStatus = Status.UNKNOWN;
14
15     this.peerPoller = new PeerPollingService(this, peersExecutor,
       blockchain, peers);
16     this.peerBroadcaster = new PeerBroadcastingService(this, peersExecutor
       );
17 }
18
19 public void startPolling(){
20     this.peerPoller.start();
21 }
22
23 public void broadcastBlock(Block block){
24     this.peerBroadcaster.broadcastBlock(block);
25 }
```

Listing 3.34: Peer class constructor and polling/broadcasting controller methods

Again, thread locking is required which is done through the use of a ReentrantReadWriteLock (lines 1 to 3). A constant is defined to hold the peers socket string (line 5) which is passed down from the peers service bean and is assigned within the constructor (line 12). A Status enum is also defined as peerStatus for the peer object (line 6) which consists of three states: UP, DOWN and UNKNOWN. The peerStatus is initially set to UNKNOWN within the constructor (line 13) as no connection would have been made at that point. Finally, PeerPollingService and PeerBroadcastingService objects are initialised (lines 15 to 16). The associated controller method for the polling service called startPolling() simply starts the thread executor (lines 19 to 21). The controller method for the broadcasting service called broadcastBlock() (lines 23 to 25) that takes in a block constructor argument which is passed in through the top level method in the peers service bean (see Listing 3.33). The getters/setters for the Peer class can be seen in Listing 3.35.

```
1  public String getPeerSocket() {
2      //Value is set on initialisation and will not change so no locking
       required
3      return this.PEER_SOCKET;
4  }
5
6  public Status getPeerStatus() {
7      //Read lock whilst getting peer status as multiple peer threads can
       read/write to this
8      readLock.lock();
9
10     try {
11         return this.peerStatus;
12     } finally {
13         readLock.unlock();
14     }
15 }
16
17 public void setPeerStatus(Status peerStatus) {
18     //Write lock whilst getting peer status as multiple peer threads can
       read/write to this
19     writeLock.lock();
20
21     try {
22         this.peerStatus = peerStatus;
23     } finally {
24         writeLock.unlock();
25     }
26 }
```

Listing 3.35: `Peer` class getters/setters

Since the `peerStatus` can be accessed and modified by multiple different threads, locking similar to what was done within the peers and blockchain service beans is applied here (lines 6 to 26). However, since the `PEER_SOCKET` variable is never modified after initialisation, no locking is required for its respective getter method (lines 1 to 4).

3.3.4.3 Peer Client

In order for the peer polling/broadcasting services to make requests and process their responses various helper methods are defined within `PeerClient` class. The class utilises the `OkHttpClient` (Square, 2019) package that provides a simple way to interface with the peer's endpoints. The `PeerClient` class can be seen in Appendix F.

Within the constructor the `OkHttpClient` is initialised with a timeout of three sections. `Request` objects for the each of the API endpoints are also initialised with the `peerSocket` that is supplied as a constructor argument (Appendix F lines 23 to 34). These `Request` objects are then used in the various methods to interface with the peer endpoints using the OkHttp `newCall()` method.

The `getPeerBlockchain()` method is used to retrieve the entire blockchain from the peer using the `GET /blockchain` endpoint (Appendix F lines 46 to 57). The JSON payload received is converted into an array list of block objects utilising the `JSONArray()` class provided by Spring, which is subsequently returned.

The `getPeerBlockchainSize()` method retrieves the peer's blockchain size from the `GET /blockchain/size` endpoint (Appendix F lines 65 to 69). All though the response is a single integer, since the JSON response body returned is in string format it needs to parsed

before being returned.

The `getPeerLastBlock()` method retrieves the last block from the peers blockchain through the `GET /blockchain/lastblock` endpoint (Appendix F lines 81 to 85). The JSON payload is converted into a block object using the `JsonBlockUtil.getBlockFromJsonObject()` method which utilised the `JSONObject` class provided by Spring.

The `getPeerHealth()` method retrieves the peers health status from the `GET /actuator/health` endpoint (Appendix F lines 94 to 98). The `JSONObject` class is utilised to extract the status from the JSON payload as a string, which is subsequently returned.

The `addBlockToPeer()` method takes in a block object as an argument and converts it into a request body utilising OkHttp's `RequestBody` class (Appendix F lines 111 to 123). The request body is then used to build a `Request` object with the `POST /blockchain/addblock` endpoint, which is then sent to the remote peer. The response retrieved would either be a success or exception message (see section 3.3.3) which is returned as a string.

The `getHealthySocketsList()` method retrieves the socket list from the peer through the `GET /peers` endpoint (Appendix F lines 133 to 154). The response retrieved is converted into a `JSONArray` object which is then filtered to only contain the sockets that have `UP` status. If the filtered string contains sockets, it is returned, otherwise an empty string is returned.

3.3.4.4 Peer Polling

The PeerPollingService is whats responsible for keeping the local node up to date with the peer. This includes the peers health, what other peers said peer has discovered, as well as the changes to the peers blockchain. The class properties, constructor method and the method used to submit the classes runnable task can be seen in Listing 3.36.

```
1  private final Blockchain blockchain;
2  private final Peers peers;
3
4  private final Peer peer;
5  private final PeerClient peerClient;
6
7  private final ScheduledThreadPoolExecutor peersExecutor;
8
9  //The delay between each poll in ms
10 private static final long POLLING_FREQUENCY = 5000;
11 private String cashedPeerSocketsList;
12
13 public PeerPollingService(Peer peer, ScheduledThreadPoolExecutor
       peersExecutor, Blockchain blockchain, Peers peers) {
14     this.blockchain = blockchain;
15     this.peers = peers;
16
17     this.peer = peer;
18     this.peerClient = new PeerClient(peer.getPeerSocket()); //Instantiate
       a new peer client from the peers socket
19
20     this.peersExecutor = peersExecutor;
21 }
22
23 private long getRandomInitialDelay(){
24     //Random delay is calculated using a random value with the range
       specified by the polling frequency
25     double delay = ThreadLocalRandom.current().nextDouble() *
       POLLING_FREQUENCY;
26
27     return Double.valueOf(Math.ceil(delay)).longValue();
28 }
29
30 public void start() {
31     peersExecutor.scheduleAtFixedRate(this, this.getRandomInitialDelay(),
       POLLING_FREQUENCY, TimeUnit.MILLISECONDS);
32 }
```

Listing 3.36: Peer class getters/setters

Both the blockchain and peers service beans are passed down into the class through the constructor (lines 14 to 15) since as explained in section 3.3.4.1, they cannot be injected through spring as the class does not run a service bean. The peer object related to the instantiated polling service is passed in through the constructor (line 17) as the status is altered during the runnable tasks execution. The peersExector that is defined and maintained within the peers service bean is also passed in through the constructor as it whats used to submit and run the threaded task. A new peer client object is initialised for the polling service using the socket retrieved from the peer object (line 18). Once all the parameters have been initialised, the start() method can be called which submits the runnable task using the scheduleAtFixedRate() method (lines 30 to 32). This method will continuously submit

the runnable task to the executor. The rate at which the tasks are submitted is defined within the `POLLING_FREQUENCY` constant (line 7), which is specified in milliseconds. A random initial delay is calculated within the `getRandomInitialDelay()` method (lines 23 to 28) which is based of the `POLLING_FREQUENCY` constant. The reason a random initial delay is required is so that when the server starts up the initial peers have their polling services staggered which helps the scheduling of the thread pool executor.

The `PeerPollingService` class implements the Java `Runnable` interface which provides a `run()` method, for which the contents of said method are executed as a threaded task for the class. A sequence diagram describing the `run()` method for the `PeerPollingService` class communicating to a healthy peer can be seen in Figure 3.13, with the full method contents being shown in Appendix G.

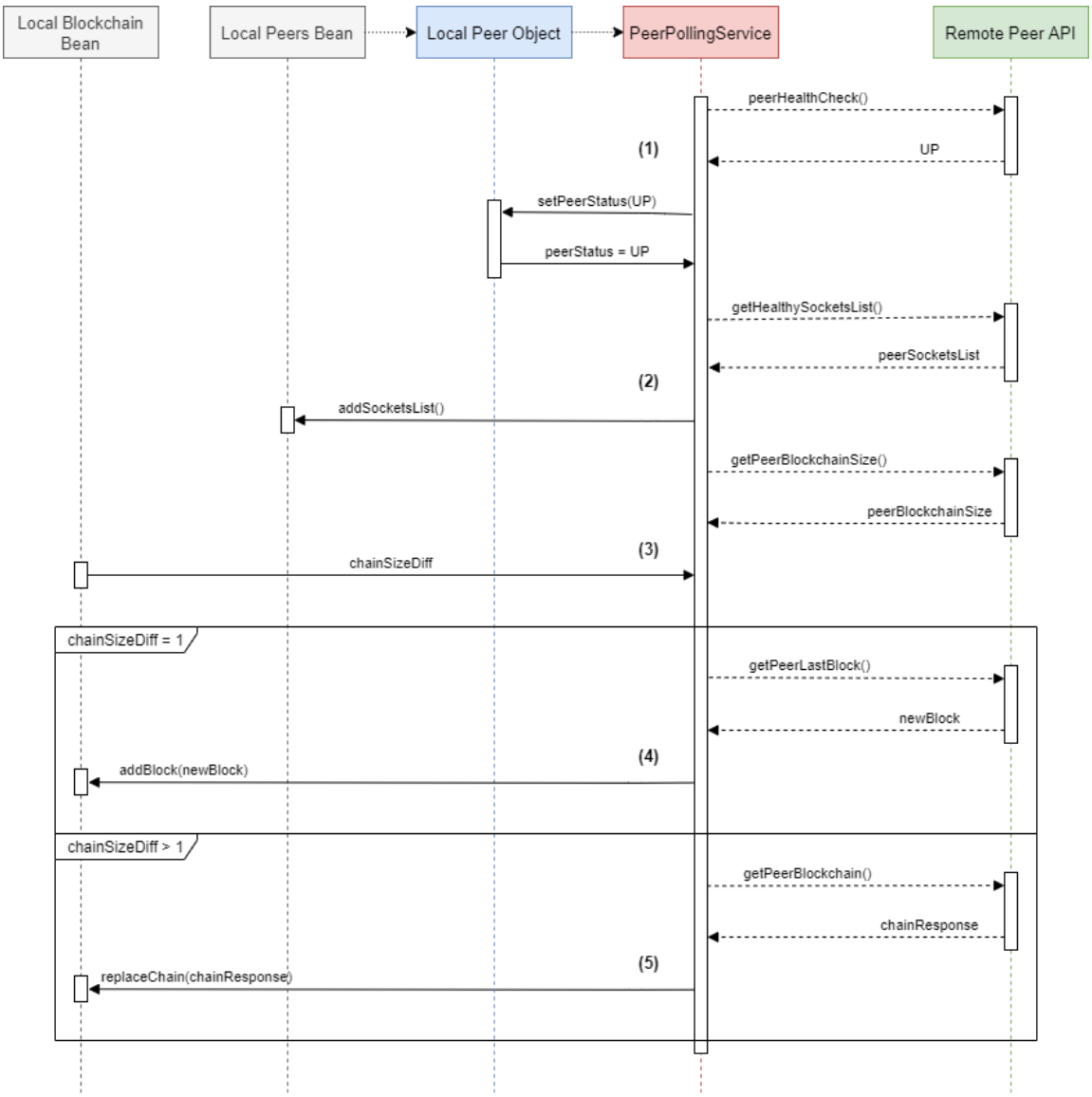


Figure 3.13: Polling service sequence diagram for a healthy peer

First, as shown in Figure 3.13 section (1), the remote peers health is checked to ensure the peer is still connectable. The local peer objects `peerStatus` property is then set based on the response, and if the peer status is `UP` the polling service will continue. The `peerHealthCheck()`

method that carries out this process can be seen in Listing 3.37.

```
/**
 * Gets the peers health status from its actuator endpoint
 * and sets its status variable for this polling services respective peer
 *
 * @throws IOException   connection exceptions
 * @throws JSONException json conversion exceptions
 */
private void peerHealthCheck() throws IOException, JSONException {
    String peerHealth = peerClient.getPeerHealth();

    if(peer.getPeerStatus() != Status.UP){
        if(peerHealth.equals("UP")){
            peer.setPeerStatus(Status.UP);
            log.info("Connection reestablished with peer: [{}] ! Setting
    peer status to UP", peer.getPeerSocket());
        }
        else{
            peer.setPeerStatus(Status.UNKNOWN);
            log.info("Peer health check returned invalid response: {}.
    Setting peer [{}] status to UNKNOWN", peerClient.getPeerHealth(), peer.
    getPeerSocket());
        }
    }
}
```

Listing 3.37: The `peerHealthCheck()` method within the `PeerPollingService` class

The peers sockets list is then retrieved using the `peerClient.getHealthySocketsList()` method and is compared against the `cashedPeerSocketsList` stored locally within the peer polling service. If there is a difference between the cashed sockets and what was retrieved by the server, new peers are available for the node to connect to. These peers are then initialised with the peers service bean through the `addSocketsList()` method, shown in Figure 3.13 section (2).

The peers blockchain size is then retrieved using the `peerClient.getPeerBlockchainSize()` method which is compared against the local beans chain size to calculate the `chainSizeDiff` variable, shown in Figure 3.13 section (3). If the `chainSizeDiff` is one, that means the remote peer has a new block which is subsequently retrieved using the `peerClient.getPeerLastBlock()`. The new block is then added to the local nodes blockchain service bean through the `addBlock()` method, shown in Figure 3.13 section (4). However, if the `chainSizeDiff` is greater than one, it means the local blockchain is more than one block behind the nodes blockchain and is therefore out of sync with the network. If this is the case, the entire blockchain is retrieved from the remote peer through the `peerClient.getPeerBlockchain()` method which is used to the replace the `chain` array in the local blockchain service bean through the `replaceChain()` method, shown in Figure 3.13 section (5).

With this, the task is complete and the thread used within the thread pool executor is freed up to be used by other tasks. After the period specified by `POLLING_FREQUENCY` the task will be resubmit to the thread pool executor to carry out the next polling cycle. If the node is not connectable, either a `SocketTimeoutException` or `ConnectException` will be thrown which is caught within the task `run()` method, and the peer objects `peerStatus` will be set to `DOWN` (Appendix G lines 58 to 63). If another exception is thrown the `peerStatus` is set to `UNKNOWN` (Appendix G lines 64 to 68).

3.3.4.5 Peer Broadcasting

When a new block is successfully added to local nodes blockchain, it needs to be sent to the nodes known peers so that the network as a whole can quickly gain consensus on the new blockchain, which allows miners to start generating the proof of work for the next block. This is done within the PeerBroadcastingService class that can be seen in Listing 3.38.

```
1  private final Peer peer;
2  private final PeerClient peerClient;
3
4  private final ScheduledThreadPoolExecutor peersExecutor;
5
6  private Block blockToBroadcast;
7
8  public PeerBroadcastingService(Peer peer, ScheduledThreadPoolExecutor
       peersExecutor) {
9      this.peer = peer;
10     this.peerClient = new PeerClient(peer.getPeerSocket()); //Instantiate
       a new peer client from the peers socket
11
12     this.peersExecutor = peersExecutor;
13 }
14
15 public void broadcastBlock(Block block) {
16     this.blockToBroadcast = block;
17     peersExecutor.submit(this);
18 }
19
20 @Override
21 public void run() {
22     //Only broadcast if the peer is UP
23     if(peer.getPeerStatus() == Status.UP){
24         try {
25             log.debug("Attempting to broadcast block: {}, to the following
       peer [{}]...", blockToBroadcast.toString(), peer.getPeerSocket());
26             peerClient.addBlockToPeer(blockToBroadcast);
27             log.debug("...Block was broadcasted successfully!");
28         } catch (SocketTimeoutException | ConnectException e) {
29             if(peer.getPeerStatus() != Status.DOWN){
30                 peer.setPeerStatus(Status.DOWN);
31                 log.info("Could not broadcast to the following peer: [{}].
       Reason: {}. Setting peer status to DOWN", peer.getPeerSocket(), e.
       getMessage());
32             }
33             log.debug("Could not broadcast to the following peer: [{}].
       Reason: {}. Peer status is: {}", peer.getPeerSocket(), e.getMessage(),
       peer.getPeerStatus());
34         } catch (Exception e) {
35             peer.setPeerStatus(Status.UNKNOWN);
36             log.debug("Could not broadcast to the following peer: [{}].
       Reason: {}. Setting peer status to UNKNOWN", peer.getPeerSocket(), e.
       getMessage());
37             e.printStackTrace();
38         }
39     }
40 }
```

Listing 3.38: The PeerBroadcastingService class Runnable implementation

The class follows a very similar format to the polling service (see 3.3.4.4) as it also implements Java's `Runnable` interface. The peer object related to the broadcasting service being instantiated is passed in through the constructor which is used to initialise a `PeerClient` object (lines 8 to 10). The thread pool executor defined within the peers service bean is also passed down through the constructor (line 12), note that this is the same executor object as is used with the polling service meaning they both share the threads within the pool.

The controller method used to submit the `run()` methods task to the thread pool executor is named `broadcastBlock()` (lines 15 to 18). This method takes in a block object which is then assigned to the `blockToBroadcast` variable, before submitting the task using the `submit()` method. This method only submits the task once unlike the `scheduleAtFixedRate()` method used to the submit the polling service task.

The `run()` method is very simple as it only needs send the `blockToBroadcast` object to the remote peer. This is done using the `peerClient.addBlockToPeer()` method (line 26) with the rest of the methods contents just being error handling to catch connection and other exceptions and set the peer objects status accordingly (lines 28 to 39). Also, its important to note that the method will only submit the block if the peer objects status is `UP` (line 23). This increases performance as if the method were to attempt a connection to an unhealthy server, a thread in the pool will be taken up for the duration of the said connection attempts timeout, for which that thread could have been used by other tasks.

3.3.5 Configuration and External Properties

Spring provides an easy and centralised way to configure the bean initialisation on server startup through the use of the `@Configuration` tag. A class tagged with such annotation instructs Spring to run said class first. Within this application the configuration class named `JrcServerConfig` handles the initialisation of the blockchain, peers and API delegate service beans, as well as the injection of external properties form outside the JAR. Said configuration class can be seen in Listing 3.39.

```
1  @Configuration
2  public class JrcServerConfig {
3      @Value("${server.address}" + ":" + "${server.port}")
4      private String NODE_SOCKET;
5
6      @Value("${peers.max}")
7      private Integer PEERS_MAX;
8
9      @Value("${peers.sockets}")
10     private String PEERS_SOCKETS;
11
12     @Bean
13     public Blockchain blockchain(){
14         // Initialise an empty blockchain instance for this node
15         return new Blockchain(new ArrayList<>());
16     }
17
18     @Bean
19     public BlockchainApiDelegateImpl blockchainApiDelegateImpl(){
20         // Initialise the blockchain API implementation for this node
21         return new BlockchainApiDelegateImpl();
22     }
23
24     @Bean
25     public Peers peers() {
26         // Initialise the peers for this node
27         return new Peers(NODE_SOCKET, PEERS_MAX, PEERS_SOCKETS);
28     }
29
30     @Bean
31     public PeersApiDelegateImpl peersApiDelegateImpl(){
32         // Initialise the peers API implementation for this node
33         return new PeersApiDelegateImpl();
34     }
35 }
```

Listing 3.39: The `PeerBroadcastingService` class `Runnable` implementation

There are three properties defined which are used to hold the external configuration data: `NODE_SOCKET`, `PEERS_MAX` and `PEERS_SOCKETS` (lines 3 to 10). These variables have the data from their respective property injected using the `@Value` tag. Each bean is then initialised by returning a new object of the beans respective class within a method of the beans name, which is tagged with the `@bean` annotation. The blockchain bean (lines 13 to 16) is initialised with an empty array list which is used for the `chain` array (see Listing 3.24). The peers bean is initialised with the external properties which provide the initial sockets list, maximum number of peers and the nodes socket (see Listing 3.31). The API delegate implementation beans require no constructor arguments so are simply initialised as is (lines 19 to 22, and 31 to 34). The external properties file used is named

application.properties as is default with Spring, and an example can be seen in Listing 3.40. Said file is placed within the same directory as the JAR application, with the --spring.config.location="file:application.properties" command being used on execution of the JAR.

```
#Node Address
server.address=82.9.184.74
server.port=8080
#Peer Socket List
peers.sockets=82.9.184.74:8081,127.0.0.1:8082,54.144.251.53:8080
#Maximimum number of peers allowed
peers.max=12
```

Listing 3.40: Example Application.properties external configuration file

3.3.6 Logging with SLF4J, Lombok and Logback

Throughout the application logging is simplified through the use of SLF4J and Lombok. The Simple Logging Facade for Java (SLF4J) provides an abstraction for the various different logging framworkds within Java, allowing the the desired framework to be plugged in at deployment time (QOS.ch, 2004). Lombok allows any a class to be annotated with a logging tag (for which @Slf4j is used within this application) which provides a static final log field without having to explicitly define it (The Project Lombok Authors, 2009). The logging framework implemented within the application is Logback which integrates with the Tomcat container and provides HTTP access log functionality (qos.ch, 2019). It also provides a simple and centralised logging configuration file, for which the implementation within the application can be seen in Listing 3.41.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
  <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
   <pattern>
     %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level &lt;%class{0}&gt; - %msg%n
   </pattern>
  </encoder>
 </appender>

 <root level="debug">
  <appender-ref ref="CONSOLE"/>
 </root>
</configuration>
```

Listing 3.41: Logback.xml logging configuration file

The format in which logs are displayed to the console is defined within the <pattern> tags (lines 5 to 7), and the logging level is defined with the <root> tags (lines 11 to 13). Within the application the pattern used displays: the exact time the log occurred; the thread the log occurred within; the logging level; the class where the log occurred and finally the log message. This pattern gives sufficient detail in order to effectively monitor and debug the application. The logging level specified will display logs of all levels below what is defined. This means if DEBUG mode is selected, all logs will be displayed, if INFO is selected, info and error logs will be displayed, and if ERROR is selected only error logs will be displayed.

3.3.7 Unit Testing and CI Pipeline

Within the application every class has an associated test class which follows the Maven standard directory structure. A UML representation overview of the test classes can be seen Appendix H, with each classes implementation being shown in Appendix I.

Testing has been carried out to be as verbose as possible with Java best practices being used throughout. A 'given-when-then' BDD testing approach is taken in order to make the test methods as clear and understandable as possible (SpecFlow, 2021), with each given, when and then segment sectioned with a comment. Overall the tests produce a code coverage of 98%, which is a measure of the proportion of production lines of code that are touched by a test, with a general industry standard of greater than 70-80% being considered acceptable (Cornett, 2013).

The unit tests are written using Junit which provides various 'assert' methods that allow for test results to be easily validated. If an assert fails, an 'orange' test failure will be displayed indicating exactly which assert failed and what it was expecting. If the test fails due to an exception being thrown (which isn't being tested) a 'red' test failure will be displayed. An orange test failure suggests there are problems with the behaviour of the class being tested, whereas a red test failure suggests there is a problem with how that class is implemented. Each test case method is annotated with a `@Test` tag which instructs the Junit runner to include said method within the test suite.

The mocking framework used is Mockito (*The Mockito Framework*, 2008), which provides a simple API that produces readable tests with clean validation errors (as opposed to other similar frameworks). Mocking is an essential technique utilised in unit testing as it allows for dependant classes and thier behaviour to be 'faked' in order to isolate the behaviour of the actual class being tested. With Mockito, the mocked objects can be part of the unit test as method calls and arguments to said mock can verified.

There are two Continous Integration (CI) pipelines implemented that can automatically run the entire test suite on an external cloud server. The two CI platforms used are CirleCI and TravisCI, which are both free and widely used CI tools. Both of the pipelines are setup to the Maven test goal for the project which runs the unit test suite. The pipelines are integrated within the GitHub repository so that every commit will cause the pipeline to run. If either pipeline fails, an email will be sent detailing the exact tests that caused the failure, which vastly improves the efficiency of development within the project as errors are quickly identified without having to continuously run the test suite manually (which takes takes up resources as well as a considerable amount of time).

3.4 Summary

In summary, this chapter has provided a detailed documentation of how all the components within this project were implemented.

First, an overview of the CryptoNight algorithm's design was provided, following the full documentation of the CryptoNightJ and CryptoNightJNI components source code. An in depth description of each class that makes up the P2P blockchain server application is then provided, in reference to its three main components: the core blockchain structure; the server API and the peer communication. Details of the server applications internal and external configuration files are given, as well as the various logging frameworks that are implemented. Finally, an overview of the unit testing frameworks and techniques that are used within the project is provided, with the full unit test suite code available in Appendix I.

Chapter 4

Results Analysis and Discussion

4.1 CryptoNight

The following section first presents the results observed through experimentation of the two Java CryptoNight solutions developed within this project, with a comparison to other available solutions (section 4.1.1). The results of optimisation testing that was carried out during development of said algorithms is also provided (section 4.1.2). Finally, a short analysis and comparison of the two solutions produced is given, with a proposal of what could possibly be the best approach going forwards (section 4.1.3).

4.1.1 Java CryptoNight Performance Testing

The two Java solutions developed within this product were performance tested against the original C++ implementation within the Monero source code (2014), as well as the following implementations in other languages: Iwaszko's pure python solution; Klinec's Python wrapper solution; Peters' pure Rust solution; Wei's Rust wrapper solution and Ilia's pure JavaScript solution. A test environment similar to that seen in Listing 3.13, `TestHashSpeed()` method was implemented for each solution, with the five hashes seen in Table 3.1 tested 1000 times each, using Eq. 3.3 to calculate each solutions respective hash rate. The tests were conducted on an Intel Core i7-10750H CPU, which has boost clock speeds of up to 5.00 GHz, with similar a load between tests (as controlled as possible). The results of the experimentation can be seen in Figure 4.1.

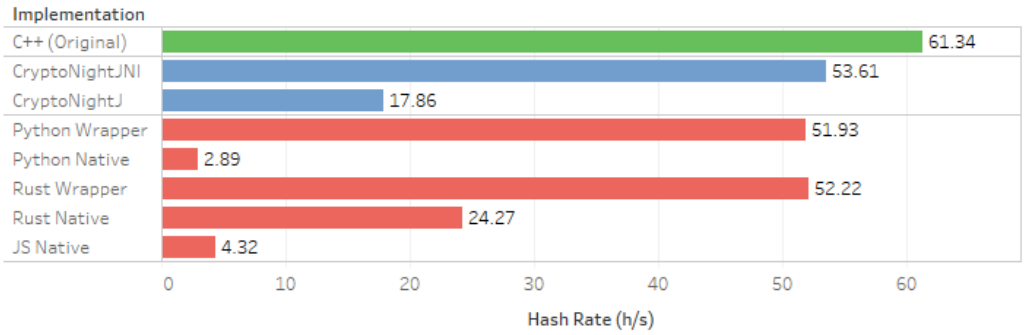


Figure 4.1: CryptoNight solutions hash rate comparison

The original C++ implementation performs the best out of all the tested solutions, with a hash rate of 61.34. This is to be expected as C++ provides a lot of powerful low level functionality in comparison to the other more high level languages. This solution is also the most used and tested implementation, and therefore it is expected to be the most optimal.

The CryptoNightJNI wrapper solution performs quite well with a hash rate of 53.61. The performance decrease in comparison to the C++ solution is to be expected as the JNI bindings as well as object creation within the test method adds a small amount of execution time to each hash. Quite similar results are observed with the Python and Rust wrapper solutions, with hash rates of 51.93 and 52.22 respectively. The CryptoNightJNI solution appears to be slightly more performant than the other wrapper solutions which could suggest that the JNI is more effective at cross language binding than other methods, however the difference in hash rate observed is too minute for this to be conclusive.

The difference in hash rate with the pure language solutions appears to be much more variable, which could be due to different levels of optimisation within each solution. The CryptoNightJ solution performs much better than the pure Python and JavaScript solutions,

with a hash rate of 17.86 in comparison to 2.89 and 4.32 respectively. However, it has a considerable performance decrease in comparison to the pure Rust solution, with a hash rate of 24.27. This suggest that further optimisation to the CryptoNightJ solution could be made as all though its usable in comparison to the Python and JavaScript pure solutions, its performance isn't ideal. However, limitations within the Java language itself might be the cause of performance loss observed, and if so further optimisation attempts could be futile.

4.1.2 Keccak Optimisation

One such optimisation attempt in regards to the Keccak component was performed. This component is initially used to produce the final state within the scratchpad initialisation stage (see section 3.2.1.1) as well as being used at the end of the algorithm within the results calculation stage (see section 3.2.1.3). Two modified Keccak solutions were tested including Melsha's (which was ultimately implemented) and Bobulous' solutions. A very similar test environment and hash rate calculation method to what was done during overall CryptoNight experimentation was used (see section 4.4.1), however, five different hashes were calculated 1,000,000 times each, as the algorithm on its own is much faster than CryptoNight. The full test repository can be found at: `https://github.com/jounaidr/Keccak-java-speedtest`, with the hash rate results shown in Table 4.1.

Table 4.1: Java Keccak solutions hash rate comparison

Melsha	Bobulous
2×10^6	4.97×10^5

4.1.3 Analysis of the Two Java Solutions

Given the performance decrease observed in the CryptoNightJ solution, it is preferable to use the CryptoNightJNI solution within both this projects server application, as well as other future Java projects. However, this introduces an issue surrounding the portability of such applications, as the CryptoNightJNI solution only includes binaries for Windows and UNIX systems, which is not ideal as portability is one of the main benefits of Java applications. Therefore a combined approach of the two solutions is proposed, where instead of throwing an exception if the operating is not supported within the JNI solution (see Listing 3.15), the CryptoNightJ solution is used. This would allow for a single Java CryptoNight solution that would be mostly optimal (as its highly likely the system will be of either Windows or UNIX) and could run entirely within the JVM if need be (albeit less optimally).

4.2 P2P Network

The following section details the integration and component testing that was carried out on the server application to verify its functionality in regards to the initial requirements. Details of a local integration test which verified the process of blocks being added to the blockchain is first given (section 4.2.1). A short explanation of how the AWS node that was used during component testing across subnets, was initially setup is then provided (section 4.2.2), before the details of each component test is documented (section 4.2.3 and 4.2.4). Finally, an explanation of the Postman client used to component test the block validation functionality is given, with an overview of the various relevant tests that were performed (section 4.2.5).

4.2.1 Local Blockchain Integration Test

To initially test that blocks are added as expected when the application runs, as well as to verify the difficulty adjusts correctly, a local integration test was created, which can be seen in Appendix J.

The test initialises a new blockchain object named `testChain` (Appendix J line 8), which then has blocks continually added to it for the time specified in the `timeToRun` property, which is default to 30 minutes (Appendix J line 3). In order to have the blocks added for the specified amount of time, a new thread is created and instantly slept for the duration specified by the `timeToRun` property before being interrupted. The test methods are then ran within a while loop with the condition set as the previously spawned thread not being interrupted (Appendix J lines 10 to 30). The time difference between each block is calculated and is output to the console using the `BlockUtil.calcBlockTimeDiff()` utility method (Appendix J lines 27 to 29).

Once the test has finished running, the console can be analysed to check whether the blocks produced approximate the set block time, and whether the difficulty adjusts correctly around said block time. After running the test for 30 minutes, the actual block time observed was between 80 and 250 seconds (with a target block time of 120 seconds), with the difficulty ranging between 9 to 13. This integration test confirmed that the difficulty adjustment and process of blocks being added to the blockchain functions correctly and as expected.

4.2.2 AWS Instance Configuration

Component testing required the use of a node outside of the local subnet testing was originally carried out on. Therefore, it was decided to test the application using an Amazon Web Services (AWS) EC2 instance running Red Hat Enterprise Linux (RHEL). The process involved generating an RSA key pair within the EC2 instance console, which is stored within the `/.ssh/authorized_keys` directory on the RHEL instance, that can then be used to SSH into the instance. Basic setup packages were installed onto the instance such as `firewalld`, and the root level SSH access was opened to allow for SFTP to setup and transfer the server files. After transferring the server application JAR and the application.properties file into the `/opt/` directory of the RHEL instance, a service file was created to run the application command through `systemd`, which is placed in the `/etc/systemd/` directory, and can be seen in Listing 4.1.

```
[Unit]
Description=Blockchain Server Application
After=syslog.target
[Service]
User=root
ExecStart=java -jar /opt/server-0.0.1-SNAPSHOT.jar
    --spring.config.location="file:/opt/application.properties" SuccessExitStatus=143
[Install]
WantedBy=multi-user.target
```

Listing 4.1: RHEL service file for server application

The service file allows the server application to be setup as a service that once enabled, will launch the JAR on server startup (using the `ExecStart` command). The required ports can also be assigned specifically to the service, with port 8080 being default for the application. A security group with the specified port also must be applied to the EC2 instance within the AWS console, and with that the server is connectable through the nodes public IP address.

4.2.3 Peer Discovery Component Testing

A manual component test with the purpose of verifying the peer discovery functionality of the server application was carried out, for which the network setup can be seen in Figure 4.2.

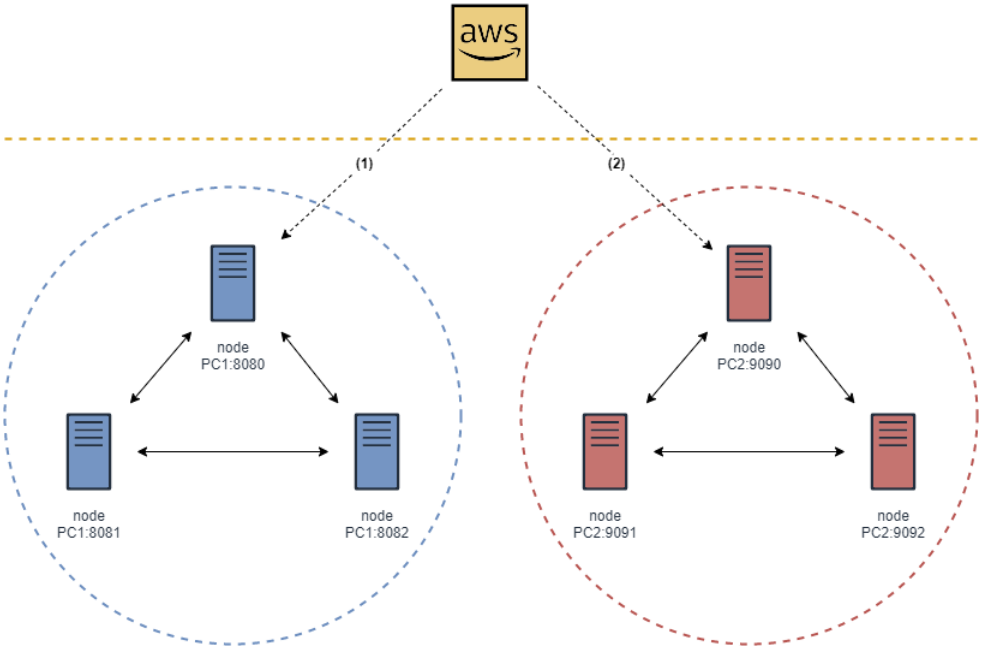


Figure 4.2: Peer discovery component test node setup

Two local clusters were setup within the same subnet (but on different machines) with each node within each cluster only being configured with peers within its cluster. The first cluster consisted of nodes on ports: 8080, 8081 and 8082, and the second cluster had nodes on ports: 9090, 9091 and 9092. Initially when all six nodes within the clusters are brought online, as expected the nodes only list peers within their cluster. This means that a node running on PC1 had no way of communicating with the nodes on PC2 and vice versa.

The AWS node is then brought up, which is configured to act as a 'bridge' between the two clusters, and knows only nodes PC1:8080 and PC2:9090 (shown by connections (1) and

(2)). After the AWS node had finished its startup sequence and had been running for a few seconds, every node within the network had every other node listed as a peer. This verified that the peer discovery functionality of the server application behaves as expected.

The test was repeated but with nodes PC1:8082 and PC2:9092 down during the whole experiment. Before connecting the AWS node to the network, the other nodes within each cluster had the respective node that was turned off listed within their peer lists with `DOWN` status as expected. After the AWS node was connected, the nodes in the PC1 cluster had discovered nodes PC2:9090 and PC2:9091, but not node PC2:9092, and vice versa with the PC2 cluster having discovered all nodes in the PC1 cluster other than PC1:8082. This verified that only nodes that are running and are connectable can be added to a nodes peer list through the peer discovery functionality.

4.2.4 Difficulty Adjustment and Block Propagation Component Testing

A manual component test with the purpose of verifying that valid blocks submitted to a node in the network propagate correctly throughout the entire network, and that the difficulty adjusts according to the networks hashing power, were carried out. The network setup for said component tests can be seen in Figure 4.3.

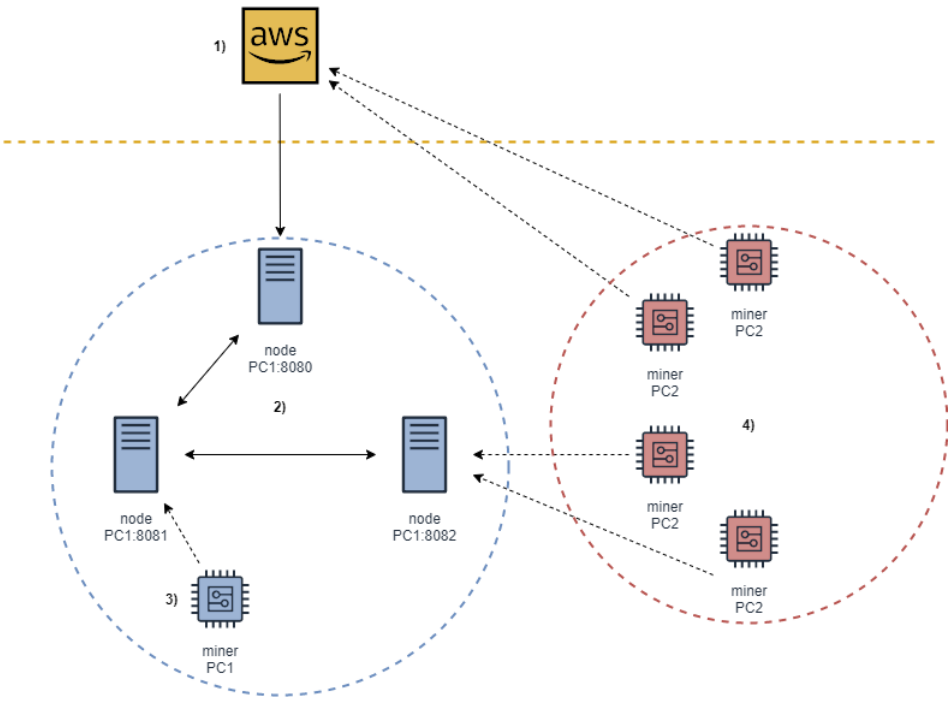


Figure 4.3: Total network hashing power component test node setup

The test network involved the AWS node, three nodes in a cluster and a miner on PC1, and four miners on PC2.

The miner application simply requests the latest block from the node its connected to, and attempts to mine the next block based on said previous block. The generated block is then submitted to the node, which will either be accepted if its the first valid next block in the network to be mined, or rejected if another valid block was found first. Either way, the miner will again request the last block from the node to repeat the mining process.

First, the AWS node is brought online and is connected to the local PC1 cluster at node PC1:8080, shown in Figure 4.3 section (2). The PC1 miner which is connected to node

PC1:8081 is then brought online, which starts producing new blocks and adding them to node PC1:8081's local blockchain. The blockchain endpoints for the rest of the nodes in the network are monitored, and its verified that the new blocks added to PC1:8081 also appear in the correct order in every other nodes blockchain. This confirms that blocks propagate through the network as expected. When the blockchain was analysed after letting the miner run for one hour, a very similar difficulty range then what was recorded during the local blockchain integration test (section 4.2.1) was also observed here, with a difficulty of between 9 and 13 generally being produced.

The second component test then involved bringing the four miners on PC2 online, two of which were connected to PC1:8082, with the other two being connected to the AWS node. PC2 has a more powerful CPU (Intel Core i9-9900K) and thus should be more effective at mining blocks in comparison to the miner at PC1. An interesting observation was made as after leaving the network with all miners running for one hour, the difficulties recorded within the blockchain were quite varied, being between 11 and 23. The fact that difficulties above 13 were seen, which was never observed when only the one miner in PC1 was connected to the network confirms that the difficulty adjusts with the hashing power of the network.

A more robust component test with a lager number of miners on different CPUs, being incrementally added to the network would provide more conclusive results. However, there wasn't sufficient hardware or available AWS nodes to carry out such a test.

4.2.5 Invalid Block Rejection

Component testing of the server applications block validation is carried out by submitting valid and invalid blocks to a running node, and verifying that the HTTP response code and message is as expected, and is equivalent to the behaviour tested within the block validation unit tests. This is done by having a single node online, with only the genesis block in its blockchain, with the next block valid block in the chain being submit through the `POST /blockchain/addblock` endpoint as a JSON payload. The node can then be restarted which will clear the blockchain (and revert it back to having just a genesis block) for which the next block can be modified and resubmit to test the various different block validation checks. The Postman client is a widely used tool in the testing of server APIs, and is whats used within this component test to submit the JSON block payloads. Figure 4.4 and Figure 4.5 show examples of a valid and invalid block being submitted to a node through Postman, with their respective response codes and messages.

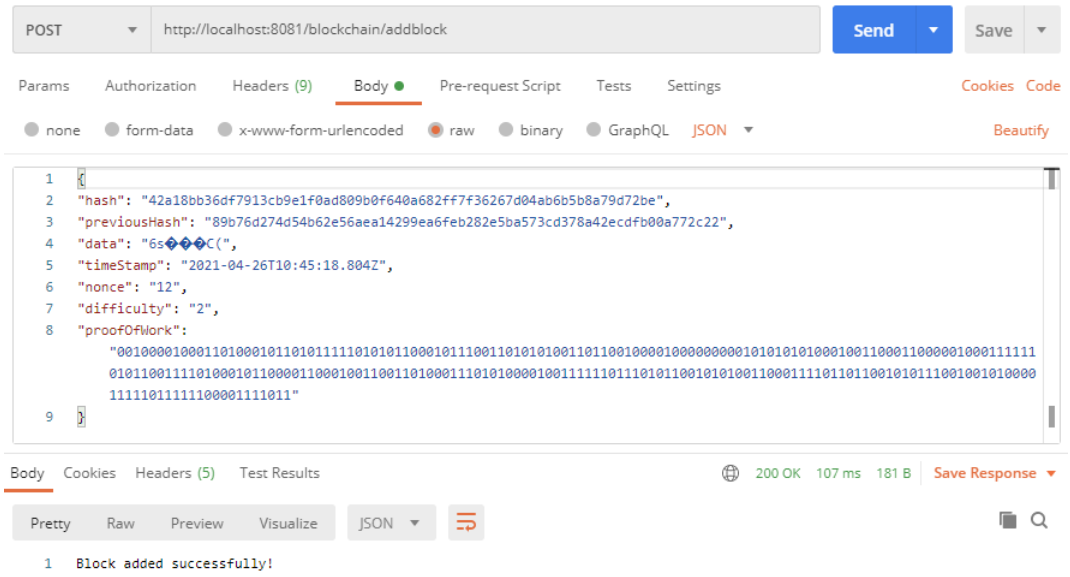


Figure 4.4: Valid JSON block submit through Postman

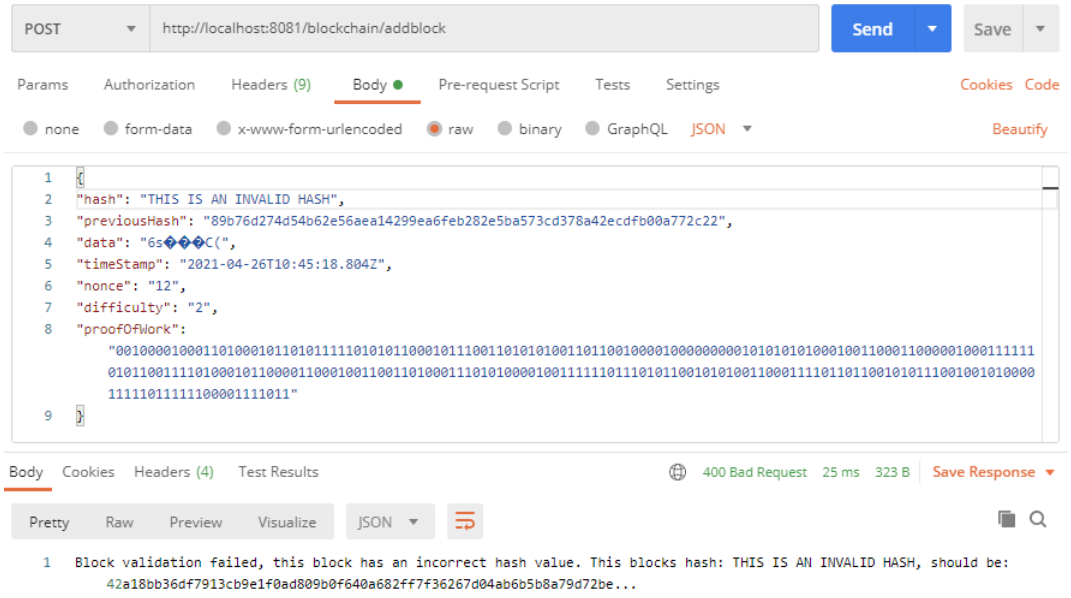


Figure 4.5: Invalid hash JSON block submit through Postman

The following tests were conduced to verify the block validation behaviour, which mirror the unit tests of the same purpose (see Appendix I.8 and Appendix I.7):

- **Submit a valid block** - *verify that response code 200 with correct success message received.*
- **Submit a block with an invalid hash** - *verify that response code 400 with the correct exception message displaying what the valid hash should be, is received.*
- **Submit a block with an invalid previous hash** - *verify that response code 400 with the correct exception message displaying what the previous blocks hash in blockchain should be, is received.*
- **Submit a block with an invalid proof of work** - *verify that response code 400 with the correct exception message is received.*
- **Submit a block with a difficulty jump** - *verify that response code 400 with the correct exception message displaying how much the difficulty has changed by, is received.*

4.3 Summary

In summary, this chapter has presented the results and observations produced within both the CryptoNight solutions performance testing, and the P2P blockchain server application integration and component testing.

The CryptoNightJNI solution produced rather good performance results, whereas the CryptoNightJ solution could benefit from further optimisations. A combined approach of the two solutions is proposed that would provide a compromise between performance and portability.

Through manual integration testing, and componenet testing with an AWS EC2 instance, the functionality of the P2P blockchain server application was verified, with details of each test case being provided.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

To conclude, through this project a Java based blockchain server application has been successfully developed, implementing the CryptoNight proof of work algorithm, as well as providing a RESTful API. All of the initial requirements have been met in some way with performance, integration and component testing carried out to provide a verifiable form of acceptance criteria.

Two Java CryptoNight solutions have been developed, being the first published implementations of said algorithm in the Java language. The first approach involved conducting a deep analysis of the available solutions published, to understand how the algorithm functions on a low level. The various methods and functions that make up the algorithm were converted into a pure Java format to produce the CryptoNightJ solution (following Java best practices and optimal functions). The second approach utilised binaries from the original C++ implementation to develop a Java wrapper solution named CryptoNightJNI. It was found through performance testing that the CryptoNightJ solution was suboptimal in comparison to other implementations. The CryptoNightJNI solution performed rather well, however it has the disadvantage of being operating system dependant. Therefore a combined approach of the two solutions was proposed, which would meet all the initial requirements related to the CryptoNight component of this project.

Modern technologies and frameworks were used to their fullest during the development of the blockchain server application. The Spring framework gave a solid foundation to the application, providing both an integrated HTTP web server environment, and easy bean initialisation/management through the use of internal and external configuration files. The OpenAPI generator integrated with Spring to produce simple RESTful API interface classes from a centralised and independent specification file. Peer broadcasting and polling services were implemented utilising Java's Runnable interface classes, with a thread pool executor being used to produce a highly efficient and performant P2P communication system, which can handle multiple peers simultaneously. Functionality allowing for peer discovery was implemented, and was verified across distant subnets through the use of an AWS EC2 instance. Comprehensive component testing involving the AWS node, as well as a Postman client was carried out, which produced results that satisfied the all the acceptance criteria set out in the initial requirements.

5.2 Future Work

The project is currently in a very good place, with verbose unit testing (achieving 98% code coverage) as well as copious amounts of documentation. The project has also been developed in a way so that it is highly modular, meaning the various components that make up the project can be easily changed, modified or replaced. However, the project as it is in its current state is really just the core of a functional blockchain P2P network, and as such there are many future works that can be carried out to both improve existing functionality, and to add additional features.

5.2.1 CryptoNight AES Optimisation

One of the main components within the CryptoNightJ solution that was observed to have caused performance loss are the AES sub-bytes, shift-rows and mix-columns method implementations. Said methods are frequently used within the algorithm, 1,310,720 times within the scratchpad initialisation stage, 524,288 times during the memory hard loop stage and

163,840 times during the results calculation stage. The current implementation works by converting the 16 byte arrays into 2D blocks before carrying out the AES methods, and then flattening the subsequent 2D array result when returning. An optimisation attempt could be made to reconfigure the sub-bytes, shift-rows and mix-columns methods to work directly on the 1D 16 byte array, thus removing the need for format conversion which potentially could make a large improvement on the algorithms performance.

5.2.2 CryptoNightJ GPU Adaption

The main purpose of the CryptoNight algorithm as mentioned is to be less efficient when ran on GPGPUs and ASICs, therefore it would be interesting to modify the CryptoNightJ solution to run on a GPGPU, and measure the performance difference observed in comparison to more basic proof of work algorithms such as SHA-2. There are various ways of adapting algorithms to run on GPUs, notably either through Nvidia's CUDA framework or through OpenCL, with Java APIs being provided in the following libraries: LWJGL, JOCL, JCUDA, Aparapi and JavaCL (Heaton, 2013).

5.2.3 Automatic Integration and Component Testing

Currently the CI pipelines implemented only run the unit test suite, however one of the main benefits of such pipelines is the ability to run component and integration tests on VMs during the pipelines execution. This could be done through the use of Jenkins with VMware integration, where after the unit test suite is run, the server application JAR is built with the new commits, which is then deployed on several VMs where automated component testing between said VMs can be carried out. However, such a pipeline would require either several AWS instances or maintaining several physical servers which would have to be brought online for every commit (or left running 24/7).

5.2.4 Streamlining the RHEL Distribution's Installation

Currently, installation is carried out through the extraction of either a zip (on windows) or tar.gz (on linux) archive which contains the server application JAR, the application.properties file and systemd service file. An RPM for RHEL systems would streamline the installation process, allowing for the service file to be placed in the correct directory and to be enabled automatically, as well as providing an initial command line interface during installation prompting the user to enter a list of peers (instead of having to modify the application.properties file directly).

5.2.5 Integrating Transactional/NFT Components

The next feature of the application that is currently in progress is a transactional component. This includes transaction objects that are submitted within the blocks data property, which allows for the general functionality of a cryptocurrency such as sender and recipient addresses and funds. Currently a separate module for the wallet component has been implemented, that allows for the creation of a public key pair used to sign transactions and submit them as transaction objects to a node. The next stage would be to integrate said component with new endpoints in the node that would allow for the wallet application to send transaction JSON payloads to the nodes transaction pool. This component is untested and very early on in development however, and as such hasn't been documented within this report.

As mentioned, since the project is highly modular, an NFT component could be implement and swapped out with the transactional components being developed. Such a component would include non-fungible-token objects rather than transaction objects, which would require some form of digital media such as an image to generate the hash. NFT blockchains are a relatively new development in the field, and such a component would add a lot of value to the project in comparison to the transactional component, which is not particularly innovative.

Chapter 6

Reflection

In reflection, this project has been an excellent learning experience, with multiple different areas within the computing discipline being involved, such as low level bitwise algorithms, networking, cloud computing and multi threaded server development. During the development of CryptoNightJ, an analysis of solutions in multiple languages was carried out, which gave insight into how the same underlying algorithm can have vastly different implementations per technology used, and how language specific techniques and functionality can effect the implementations overall efficacy. The use of agile techniques, such as a Kanban board (with GitHub projects) helped to keep track of, and prioritise tasks as the project grew larger. The extensive documentation and testing that was produced throughout the project allows for future works to be implemented with ease, even after large breaks in development.

The project was not without it's problems however, where many times a different approach to a specific problem had to be taken in order to make progress. One such situation was when the distributed Java framework, Atomix, was trialed as a P2P component within the server application. Despite seeming promising prior to implementation, it turned out that the framework was still very early on in development with incomplete documentation, and its solution only worked within the same subnet which did not satisfy the requirements for this application. There was also an issue surrounding the setup of the AWS instance and its security exceptions, which occupied a lot of project time that could have been spent on further project development. When initially developing the CryptoNgihtJ solution it quickly became clear the algorithm was highly complex, and in combination with the fact that there is very little documentation meant a larger amount of time was allocated to this component of the project than what was originally planned. Because of this, it was decided to temporarily pause development on the CryptoNightJ component and instead focus on the CryptoNightJNI solution and P2P network implementation. Near the end of the project focus was returned to the CryptoNightJ solution as opposed to progressing further with the transactional components of the server application. The reason for this was that said transnational components did not involve any innovation or originality despite requiring a lot of code to be implemented. Given this, and the restrictions placed on the project report word count and demonstration time limit, it made much more sense to complete and document the Java CryptoNight solutions fully within the remaining time, as these are the main contributions to the field this project provides. It is also more desirable to have all the components implemented within the server application at a high standard, with best coding practices as well as sufficient testing/code-coverage and documentation, which was not possible to do with the transactional components within the deadline. Therefore, releasing these components as future features will greatly increase the quality of the project as a whole.

The initial scope for the project that was setout in the PID was much too large. There were plans for both windows and web clients to be included in the first release, however it was quickly realised that these additional features did not add much value to the project, and therefore should be implemented at a later date. The PID was written before much of the initial research had taken place, and its scope did not account for other commitments during the projects development. In hindsight, the PID could potentially have been written with a smaller scope including only the core functionality, however it was always planned for this to be large project that would span multiple releases, and said documentation was mainly written for approval.

References

Barker, E., Barker, W., Burr, W., Polk, W. and Smid, M. (2007), 'Nist special publication 800-57', *NIST Special publication* **800**(57), 1–142.

Bertoni, G., Daemen, J., Peeters, M. and Assche, G. (2007), 'Sponge functions', *ECRYPT Hash Workshop 2007* .

Bobulous (2017), 'Bobulous / Cryptography'.
URL: *https://gitlab.com/Bobulous/Cryptography*

Chang, S.-j., Perlner, R., Burr, W. E., Turan, M. S., Kelsey, J. M., Paul, S. and Bassham, L. E. (2012), 'Third-round report of the sha-3 cryptographic hash algorithm competition', *NIST Interagency Report* **7896**, 121.

Chen, R. (2007), 'AESAlgorithm'.
URL: *https://gist.github.com/asika32764/224c176d89289b80e318*

Cornett, S. (2013), 'Minimum Acceptable Code Coverage'.
URL: *https://www.bullseye.com/minimum.html*

cryptoexplained (2018), 'cryptoexplained/hash-algorithms'.
URL: *https://github.com/cryptoexplained/hash-algorithms*

Dapper Labs (n.d.), 'CryptoKitties — Collect and breed digital cats!'.
URL: *https://www.cryptokitties.co/*

Fielding, R. and Reschke, J. (2014), 'Hypertext transfer protocol (http/1.1): Semantics and content'.

Fielding, R. T. (2000), *Architectural styles and the design of network-based software architectures*, Vol. 7, University of California, Irvine Irvine.

Fuhrmann, S. (2014), 'Saphir Hash'.
URL: *https://github.com/sfuhrm/saphir-hash*

Haber, S. and Stornetta, W. S. (1991), 'How to time-stamp a digital document', *Journal of Cryptology* **3**(2), 99–111.
URL: *https://doi.org/10.1007/BF00196791*

Heaton, J. (2013), 'GPU Programming in Java Using OpenCL, Part 1'.
URL: *https://www.youtube.com/watch?v=4q9fPOI-x80ab*$_c hannel = JeffHeaton$

Hooks, I. (1994), Writing good requirements, *in* 'INCOSE International Symposium', Vol. 4, Wiley Online Library, pp. 1247–1253.

Ilia, R. (2019), 'codefrom/cryptonight-js'.
URL: *https://github.com/codefrom/cryptonight-js*

Iwaszko, T. (n.d.), 'wkta/ReadableCryptoMiner'.
URL: *https://github.com/wkta/ReadableCryptoMiner*

Klinec, D. (2018), 'ph4r05/py-cryptonight'.
URL: *https://github.com/ph4r05/py-cryptonight*

Konst, S. (2000), *Sichere Log-Dateien auf Grundlage kryptographisch verketteter Einträge*, Braunschweig. Zugleich: Technische Universität Braunschweig, Diplomarbeit, 2000.
URL: *https://publikationsserver.tu-braunschweig.de/receive/dbbs*$_m ods_0 0064933$

Kramer, M. (2021), 'Non-Fungible Tokens (NFT): Beginner's Guide'.
URL: *https://decrypt.co/resources/non-fungible-tokens-nfts-explained-guide-learn-blockchain*

Lee, C. (2011), 'litecoin-project/litecoin'.
URL: *https://github.com/litecoin-project/litecoin*

Martin, J. (1983), *Managing the data base environment*, Prentice Hall PTR.

Melsha, J. R. (2016), 'jrmelsha/keccak'.
URL: *https://github.com/jrmelsha/keccak*

monero-project/monero (2014).
URL: *https://github.com/monero-project/monero*

Nakamoto, S. (2008), Bitcoin: A Peer-to-Peer Electronic Cash System, Technical report.
URL: *https://bitcoin.org/bitcoin.pdf*

netindev (2018), 'netindev/drill'.
URL: *https://github.com/netindev/drill*

NIST-FIPS (2001), 'Announcing the advanced encryption standard (aes)', *Federal Information Processing Standards Publication* **197**(1-51), 3–3.

OpenAPI (2015), 'OpenAPITools/openapi-generator'.
URL: *https://github.com/OpenAPITools/openapi-generator*

Peters, B. (2019), 'bertptrs/cryptonight-hash'.
URL: *https://github.com/bertptrs/cryptonight-hash*

Principles of Bitcoin (2017).
URL: *https://en.bitcoin.it/wiki/Principles*$_o f_B itcoin$

QOS.ch (2004), 'SLF4J'.
URL: *http://www.slf4j.org/index.html*

qos.ch (2019), 'Logback'.
URL: *http://logback.qos.ch/*

Sayeed, S. and Marco-Gisbert, H. (2019), 'Assessing blockchain consensus and security mechanisms against the 51% attack', *Applied Sciences* **9**(9), 1788.

Seigen, Jameson, M., Nieminen, T., Neocortex and Juarez, A. M. (2013), 'CRYPTONOTE STANDARD 008'.
URL: *https://cryptonote.org/cns/cns008.txt*

SpecFlow (2021), 'Given-When-Then'.
URL: *https://specflow.org/bdd/given-when-then/*

Square, I. (2019), 'OkHttpClient'.
URL: *https://square.github.io/okhttp/*

Stepanian, L., Brown, A., Kielstra, A., Koblents, G. and Stoodley, K. (2005), Inlining java native calls at runtime, pp. 121–131.

STMicroelectronics, 2NXP Semiconductors, Bertoni, G., Daemen, J., Peeters, M. and Van Assche, G. (2008), Keccak specifications, Technical Report 1 , Peeters2 and 1 1STMicroelectronics 2NXP Semiconductors.
URL: *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.419.7204rep=rep1type=pdf*

Szabo, N. (2008), 'Bit Gold'.
URL: *https://unenumerated.blogspot.com/2005/12/bit-gold.html*

The Legion of the Bouncy Castle (2000), 'The Legion of the Bouncy Castle Java Cryptography APIs'.
URL: *https://www.bouncycastle.org/java.html*

The Project Lombok Authors (2009), '@Log (and friends)'.
URL: *https://projectlombok.org/features/log*

The Mockito Framework (2008).
URL: *https://site.mockito.org/*

Van Saberhagen, N. (2013), 'CryptoNote v 2.0'.
URL: *https://cryptonote.org/whitepaper.pdf*

Wagner, L. (2021), 'Basic Intro to Elliptic Curve Cryptography'.
URL: *https://qvault.io/cryptography/very-basic-intro-to-elliptic-curve-cryptography*

Wei, T. (2018), 'Team-Hycon/cryptonight-rs'.
URL: *https://github.com/Team-Hycon/cryptonight-rs*

Yi-Cyuan, C. (2015), 'emn178/js-sha3'.
URL: *https://github.com/emn178/js-sha3*

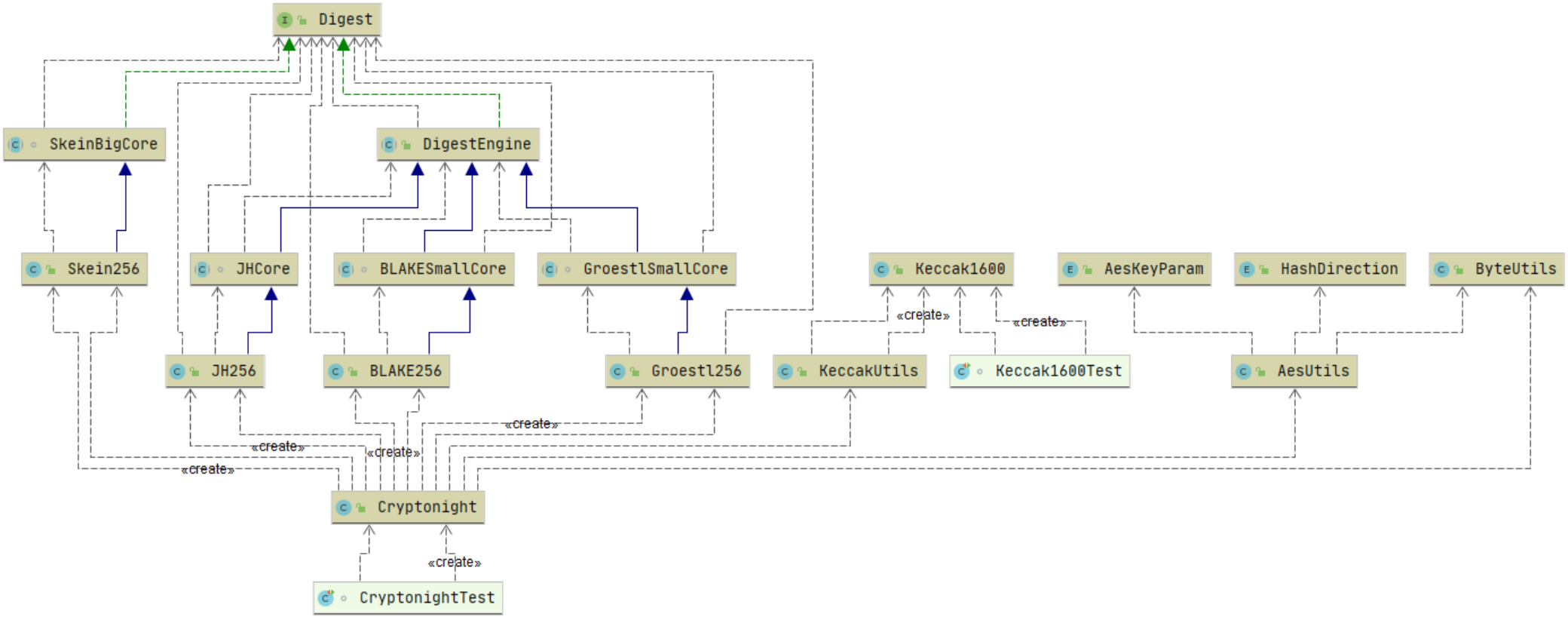
Appendix A

AES Substitution Matrix (S-box)

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

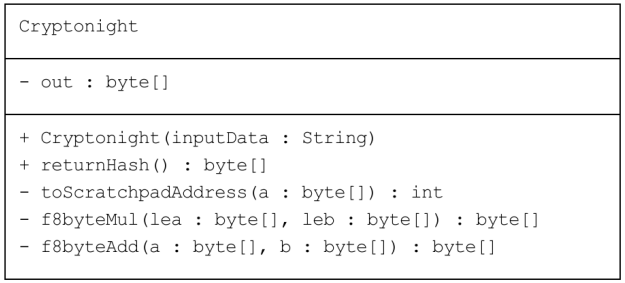
Appendix B

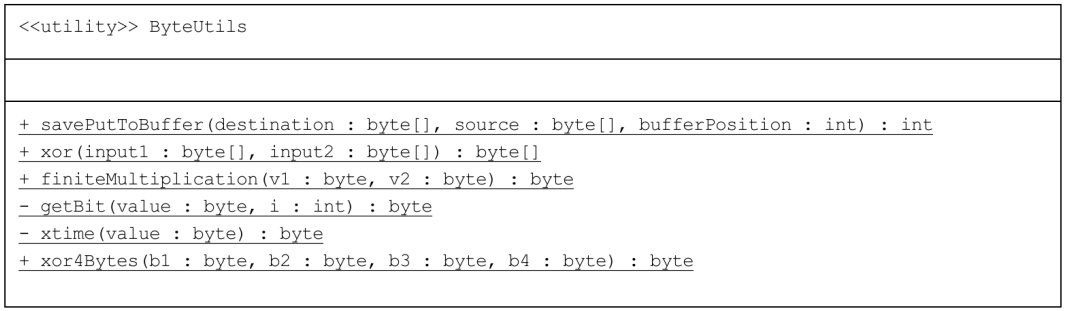
CryptoNightJ Full Class Dependency Diagram

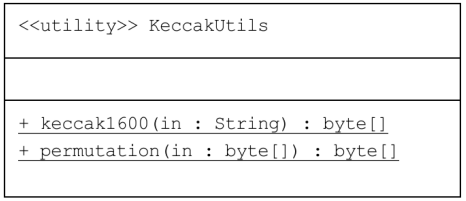

Digest
SkeinBigCore
DigestEngine
Skein256
JHCore
BLAKESmallCore
GroestlSmallCore
Keccak1600
AesKeyParam
HashDirection
ByteUtils
JH256
BLAKE256
Groestl256
KeccakUtils
Keccak1600Test
AesUtils
Cryptonight
CryptonightTest
«create»
«create»
«create»
«create»
«create»
«create»
«create»
«create»

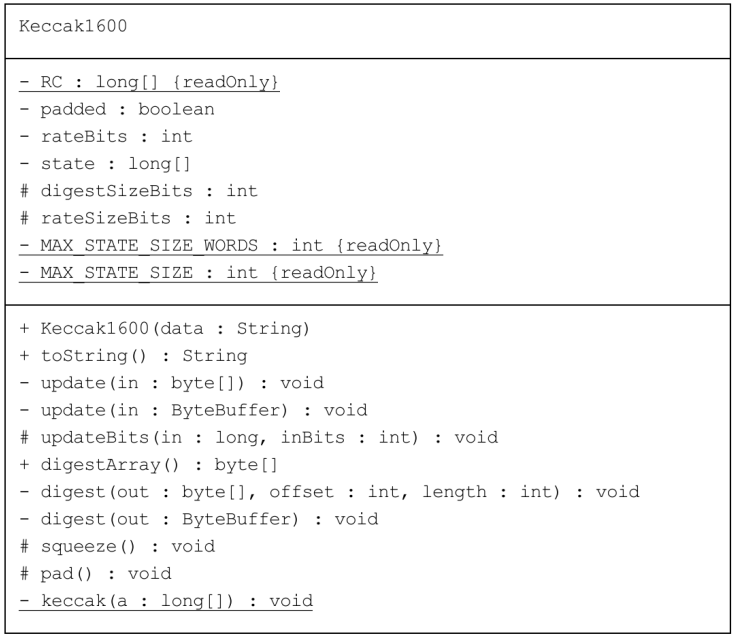
Appendix C

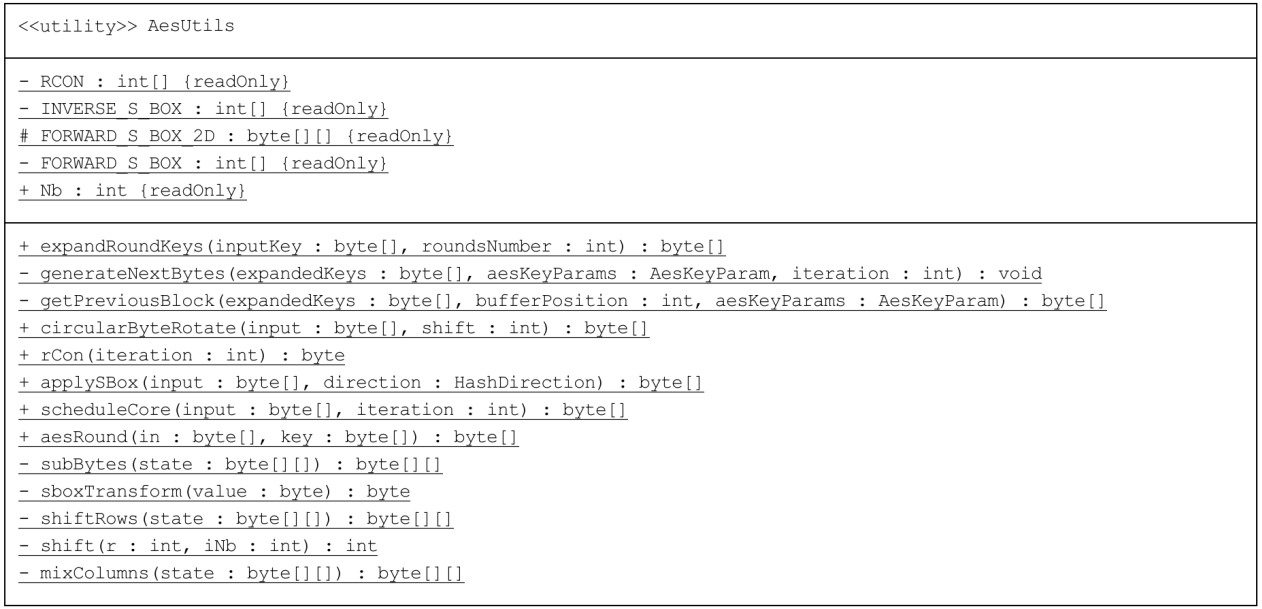
CryptoNightJ UML Class Diagrams


Cryptonight
- out : byte[]
+ Cryptonight(inputData : String)
+ returnHash() : byte[]
- toScratchpadAddress(a : byte[]) : int
- f8byteMul(lea : byte[], leb : byte[]) : byte[]
- f8byteAdd(a : byte[], b : byte[]) : byte[]


<<utility>> ByteUtils
+ savePutToBuffer(destination : byte[], source : byte[], bufferPosition : int) : int
+ xor(input1 : byte[], input2 : byte[]) : byte[]
+ finiteMultiplication(v1 : byte, v2 : byte) : byte
- getBit(value : byte, i : int) : byte
- xtime(value : byte) : byte
+ xor4Bytes(b1 : byte, b2 : byte, b3 : byte, b4 : byte) : byte


<<utility>> KeccakUtils
+ keccak1600(in : String) : byte[]
+ permutation(in : byte[]) : byte[]


Keccak1600
- RC : long[] {readOnly}
- padded : boolean
- rateBits : int
- state : long[]
digestSizeBits : int
rateSizeBits : int
- MAX_STATE_SIZE_WORDS : int {readOnly}
- MAX_STATE_SIZE : int {readOnly}
+ Keccak1600(data : String)
+ toString() : String
- update(in : byte[]) : void
- update(in : ByteBuffer) : void
updateBits(in : long, inBits : int) : void
+ digestArray() : byte[]
- digest(out : byte[], offset : int, length : int) : void
- digest(out : ByteBuffer) : void
squeeze() : void
pad() : void
- keccak(a : long[]) : void


<<utility>> AesUtils
- RCON : int[] {readOnly}
- INVERSE_S_BOX : int[] {readOnly}
FORWARD_S_BOX_2D : byte[][] {readOnly}
- FORWARD_S_BOX : int[] {readOnly}
+ Nb : int {readOnly}
+ expandRoundKeys(inputKey : byte[], roundsNumber : int) : byte[]
- generateNextBytes(expandedKeys : byte[], aesKeyParams : AesKeyParam, iteration : int) : void
- getPreviousBlock(expandedKeys : byte[], bufferPosition : int, aesKeyParams : AesKeyParam) : byte[]
+ circularByteRotate(input : byte[], shift : int) : byte[]
+ rCon(iteration : int) : byte
+ applySBox(input : byte[], direction : HashDirection) : byte[]
+ scheduleCore(input : byte[], iteration : int) : byte[]
+ aesRound(in : byte[], key : byte[]) : byte[]
- subBytes(state : byte[][]) : byte[][]
- sboxTransform(value : byte) : byte
- shiftRows(state : byte[][]) : byte[][]
- shift(r : int, iNb : int) : int
- mixColumns(state : byte[][]) : byte[][]

Appendix D

Blockchain Server UML Class Diagrams


Block
~ log : Logger
- proofOfWork : String
- difficulty : String
- nonce : String
- timeStamp : String
- data : String
- previousHash : String
- hash : String
- GENESIS_PROOF_OF_WORK : String {readOnly}
- GENESIS_DIFFICULTY : String {readOnly}
- GENESIS_NONCE : String {readOnly}
- GENESIS_TIME_STAMP : String {readOnly}
- GENESIS_DATA : String {readOnly}
- GENESIS_PREVIOUS_HASH : String {readOnly}
- MINE_RATE : long {readOnly}
- setProofOfWork(proofOfWork : String) : void
+ getProofOfWork() : String
- setDifficulty(difficulty : String) : void
+ getDifficulty() : String
- setNonce(nonce : String) : void
+ getNonce() : String
- setTimeStamp(timeStamp : String) : void
+ getTimeStamp() : String
- setData(data : String) : void
+ getData() : String
- setPreviousHash(previousHash : String) : void
+ getPreviousHash() : String
- setHash(hash : String) : void
+ getHash() : String
+ toString() : String
+ validateBlock(previousBlock : Block) : void
+ isProofOfWorkValid() : Boolean
+ generateHash() : String
- adjustDifficulty(previousBlock : Block) : void
+ mineBlock(previousBlock : Block, data : String) : Block
+ genesis() : Block
+ Block(hash : String, previousHash : String, data : String, timeStamp : String, nonce : String, difficulty : String, proofOfWork : String)
+ Block()
Peer
- peerBroadcaster : PeerBroadcastingService {readOnly}
- peerPoller : PeerPollingService {readOnly}
- peerStatus : Status
- PEER_SOCKET : String {readOnly}
- rwLock : ReadWriteLock {readOnly}
+ setPeerStatus(peerStatus : Status) : void
+ getPeerStatus() : Status
+ getPeerSocket() : String
+ broadcastBlock(block : Block) : void
+ startPolling() : void
+ Peer(peerSocket : String, peersExecutor : ScheduledThreadPoolExecutor, blockchain : Blockchain, peers : Peers)
<<enumeration>> Status
UNKNOWN
DOWN
UP
Peers
~ log : Logger
- peerList : ArrayList<Peer> {readOnly}
- poolSize : int
- peersExecutor : ScheduledThreadPoolExecutor {readOnly}
- MAX_PEERS : int {readOnly}
- NODE_SOCKET : String {readOnly}
- rwLock : ReadWriteLock {readOnly}
- blockchain : Blockchain
+ getPeerList() : ArrayList<Peer>
+ broadcastBlockToPeers(block : Block) : void
- isPeerKnown(peerSocket : String) : boolean
- isSocketValid(peerSocket : String) : boolean
- addPeer(peerSocket : String) : void
+ addSocketsList(socketsList : String) : void
+ Peers(nodeSocket : String, maxPeers : int, socketsList : String, blockchain : Blockchain)
Blockchain
~ log : Logger
- rwLock : ReadWriteLock
- chain : List<Block>
~ peers : Peers
- setChain(newChain : List<Block>) : void
+ getChain() : List<Block>
+ getLastBlock() : Block
+ isChainValid() : boolean
+ replaceChain(newBlockchain : Blockchain) : void
+ addBlock(newBlock : Block) : void
+ Blockchain(chain : List<Block>)
PeersApiDelegateImpl
- peers : Peers
+ getSocketsList() : ResponseEntity<List<PeerModel>>
<<interface>> PeersApiDelegate
getSocketsList() : ResponseEntity<List<PeerModel>>
getRequest() : Optional<NativeWebRequest>
BlockchainApiDelegateImpl
- blockchain : Blockchain
+ addBlock(newBlock : BlockModel) : ResponseEntity<Object>
+ getLastBlock() : ResponseEntity<BlockModel>
+ getBlockchainSize() : ResponseEntity<Integer>
+ getBlockchain() : ResponseEntity<List<BlockModel>>
<<interface>> BlockchainApiDelegate
getLastBlock() : ResponseEntity<BlockModel>
getBlockchainSize() : ResponseEntity<Integer>
getBlockchain() : ResponseEntity<List<BlockModel>>
addBlock(blockModel : BlockModel) : ResponseEntity<Object>
getRequest() : Optional<NativeWebRequest>
PeerBroadcastingService
~ log : Logger
- blockToBroadcast : Block
- peersExecutor : ScheduledThreadPoolExecutor {readOnly}
- peerClient : PeerClient {readOnly}
- peer : Peer {readOnly}
+ run() : void
+ broadcastBlock(block : Block) : void
+ PeerBroadcastingService(peer : Peer, peersExecutor : ScheduledThreadPoolExecutor)
JrcServerConfig
~ blockchain : Blockchain
- PEERS_SOCKETS : String
- PEERS_MAX : Integer
- NODE_SOCKET : String
+ peersApiDelegateImpl() : PeersApiDelegateImpl
+ peers() : Peers
+ blockchainApiDelegateImpl() : BlockchainApiDelegateImpl
+ blockchain() : Blockchain
java.lang.Runnable
PeerPollingService
~ log : Logger
- cashedPeerSocketsList : String
- POLLING_FREQUENCY : long {readOnly}
- peersExecutor : ScheduledThreadPoolExecutor {readOnly}
- peerClient : PeerClient {readOnly}
- peer : Peer {readOnly}
- peers : Peers
- blockchain : Blockchain
+ run() : void
- peerHealthCheck() : void
+ start() : void
- getRandomInitialDelay() : long
+ PeerPollingService(peer : Peer, peersExecutor : ScheduledThreadPoolExecutor, blockchain : Blockchain, peers : Peers)
PeerClient
- healthGetRequest : Request {readOnly}
- peersGetRequest : Request {readOnly}
- blockchainLastBlockGetRequest : Request {readOnly}
- blockchainSizeGetRequest : Request {readOnly}
- blockchainGetRequest : Request {readOnly}
- client : OkHttpClient {readOnly}
- peerSocket : String {readOnly}
+ getHealthySocketsList() : String
+ addBlockToPeer(block : Block) : String
+ getPeerHealth() : String
+ getPeerLastBlock() : Block
+ getPeerBlockchainSize() : int
+ getPeerBlockchain() : ArrayList<Block>
+ PeerClient(peerSocket : String)
<<utility>> BlockModelUtil
+ getBlockFromModel(blockModel : BlockModel) : Block
+ getBlockAsModel(block : Block) : BlockModel
<<utility>> PeerModelUtil
+ getPeerAsModel(peer : Peer) : PeerModel
<<utility>> JsonBlockUtil
+ getJsonObjectFromBlock(block : Block) : JSONObject
+ getBlockFromJsonObject(jsonBlock : JSONObject) : Block
<<utility>> BlockUtil
+ calcBlockTimeDiff(currentBlockTs : String, previousBlockTs : String) : long
+ getBinaryString(input : byte[]) : String
+ getByteArrayLeadingZeros(input : byte[]) : int

Appendix E

OpenAPI Specification File in YAML Format

```yaml
openapi: 3.0.2

############################################################
#                    API Information                       #
############################################################
info:
  title: jrc-node server api spec
  description: "OpenAPI specification for the blockchain server backend for jrc-
   node"
  license:
    name: MIT
    url: https://opensource.org/licenses/mit-license.php
  version: 0.0.1-SNAPSHOT
externalDocs:
  description: Full Documentation
  url: https://github.com/jounaidr/jrc-node

############################################################
#                API Endpoint Definitions                  #
############################################################
paths:
### Blockchain ###
  /blockchain:
    get:
      summary: Get the full blockchain
      operationId: getBlockchain
      responses:
        200:
          description: successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Blockchain'

  /blockchain/lastblock:
    get:
      summary: Get the last block in the blockchain
      operationId: getLastBlock
      responses:
        200:
          description: successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/BlockModel'




  /blockchain/addblock:
    post:
      summary: Attempt to add a new block to the blockchain
      operationId: addBlock
      responses:
        200:
          description: successful operation
          content:
            application/json:
              schema:
                type: object
        400:
```

```
          description: block is not valid
          content:
            application/json:
              schema:
                type: object
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/BlockModel'
        description: Block to be added
        required: true

  /blockchain/size:
    get:
      summary: Get the blockchain length
      operationId: getBlockchainSize
      responses:
        200:
          description: successful operation
          content:
            application/json:
              schema:
                type: integer

### Peers ###
  /peers:
    get:
      summary: Get the nodes socket list
      operationId: getSocketsList
      responses:
        200:
          description: successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Peers'

##########################################################
#                    Model Definitions                   #
##########################################################
components:
  schemas:
    Blockchain:
      type: array
      items:
        $ref: '#/components/schemas/BlockModel'

    BlockModel:
      type: object
      properties:
        hash:
          type: string
        previousHash:
          type: string
        data:
          type: string
        timeStamp:
          type: string
        nonce:
          type: string
        difficulty:
```

```
        type: string
      proofOfWork:
        type: string

  Peers:
    type: array
    items:
      $ref: '#/components/schemas/PeerModel'

  PeerModel:
    type: object
    properties:
      peerSocket:
        type: string
      peerStatus:
        type: string
```

Appendix F

Peer Client Class

```java
public class PeerClient {
    private final String peerSocket;
    private final OkHttpClient client;

    private final Request blockchainGetRequest;
    private final Request blockchainSizeGetRequest;
    private final Request blockchainLastBlockGetRequest;

    private final Request peersGetRequest;

    private final Request healthGetRequest;

    /**
     * Instantiates a new Peer client for a given peerSocket,
     * and initialises the get endpoint requests for said peerSocket.
     *
     * The peer client is used to interface with the peer, and a new
     * peer client is instantiated per thread, so thread safety is not
     * necessary...
     *
     * @param peerSocket the socket for which the peer client will be initialised for
     */
    public PeerClient(String peerSocket) {
        this.peerSocket = peerSocket;
        this.client = new OkHttpClient.Builder().readTimeout(3, TimeUnit.SECONDS).build();

        this.blockchainGetRequest = new Request.Builder().url(String.format("http://%s/blockchain", peerSocket)).build();
        this.blockchainSizeGetRequest = new Request.Builder().url(String.format("http://%s/blockchain/size", peerSocket)).build();
        this.blockchainLastBlockGetRequest = new Request.Builder().url(String.format("http://%s/blockchain/lastblock", peerSocket)).build();

        this.peersGetRequest = new Request.Builder().url(String.format("http://%s/peers", peerSocket)).build();

        this.healthGetRequest = new Request.Builder().url(String.format("http://%s/actuator/health", peerSocket)).build();
    }

    /**
     * Gets the peers blockchain.
     *
     * Each json object returned from the peers blockchain
     * is converted into a Block object using ...peer.util.JsonBlockUtil
     *
     * @return the peers blockchain
     * @throws IOException   connection exceptions
     * @throws JSONException json conversion exceptions
     */
    public ArrayList<Block> getPeerBlockchain() throws IOException, JSONException {
        ArrayList<Block> chainResponse = new ArrayList<>();

        Response blockchainResponse = client.newCall(blockchainGetRequest).execute();
        JSONArray jsonResponse = new JSONArray(blockchainResponse.body().string());

        for (int i = 0; i < jsonResponse.length(); i++) {
          chainResponse.add(JsonBlockUtil.getBlockFromJsonObject(jsonResponse.getJSONObject(i)));
        }

        return chainResponse;
    }

    /**
     * Gets the peers blockchain size.
     *
     * @return the peers blockchain size
     * @throws IOException connection exceptions
     */
    public int getPeerBlockchainSize() throws IOException {
        Response blockchainSizeResponse = client.newCall(blockchainSizeGetRequest).execute();

        return Integer.parseInt(blockchainSizeResponse.body().string());
    }

```

```
    /**
     * Gets the peers blockchains last block.
     *
     * The json object returned is converted into a Block object
     * using ...peer.util.JsonBlockUtil
     *
     * @return the peers blockchains last block
     * @throws IOException   connection exceptions
     * @throws JSONException json conversion exceptions
     */
    public Block getPeerLastBlock() throws IOException, JSONException {
        Response blockchainLastBlockResponse = client.newCall(blockchainLastBlockGetRequest).
    execute();

        return JsonBlockUtil.getBlockFromJsonObject(new JSONObject(blockchainLastBlockResponse.
    body().string()));
    }

    /**
     * Gets the peers health status.
     *
     * @return the peers health status
     * @throws IOException   connection exceptions
     * @throws JSONException json conversion exceptions
     */
    public String getPeerHealth() throws IOException, JSONException {
        Response peerHealthResponse = client.newCall(healthGetRequest).execute();

        return new JSONObject(peerHealthResponse.body().string()).getString("status");
    }

    /**
     * Submits the provided Block object to the peer.
     *
     * The block object parameter supplied is converted into json
     * format using ...peer.util.JsonBlockUtil
     *
     * @param block the block to be submitted
     * @return the return message, will return a success or error message
     * @throws IOException   connection exceptions
     * @throws JSONException json conversion exceptions
     */
    public String addBlockToPeer(Block block) throws JSONException, IOException {
        JSONObject jsonBlock = JsonBlockUtil.getJsonObjectFromBlock(block);
        RequestBody body = RequestBody.create(jsonBlock.toString(), MediaType.parse("application/
    json; charset=utf-8"));

        Request blockchainAddBlockRequest = new Request.Builder()
                .url(String.format("http://%s/blockchain/addblock", peerSocket))
                .post(body)
                .build();

        Response addBlockResponse = client.newCall(blockchainAddBlockRequest).execute();

        return addBlockResponse.body().string();
    }

    /**
     * Gets the peers socket list and builds a comma separated
     * string containing only peers with health status UP
     *
     * @return the healthy sockets list
     * @throws IOException   connection exceptions
     * @throws JSONException json conversion exceptions
     */
    public String getHealthySocketsList() throws IOException, JSONException {
        StringBuilder socketsListResponse = new StringBuilder();

        Response peersResponse = client.newCall(peersGetRequest).execute();
        JSONArray jsonResponse = new JSONArray(peersResponse.body().string());

        for (int i = 0; i < jsonResponse.length(); i++) {
            //Only get the sockets with that have status UP
            if(jsonResponse.getJSONObject(i).getString("peerStatus").equals("UP")){
                socketsListResponse.append(jsonResponse.getJSONObject(i).getString("peerSocket"))
    ;
```

```
                    socketsListResponse.append(",");
                }
            }

            if(!(socketsListResponse.toString().equals(""))){
                return socketsListResponse.substring(0, socketsListResponse.length() - 1);
            }
            else{
                return "";
            }
        }
}
```

Appendix G

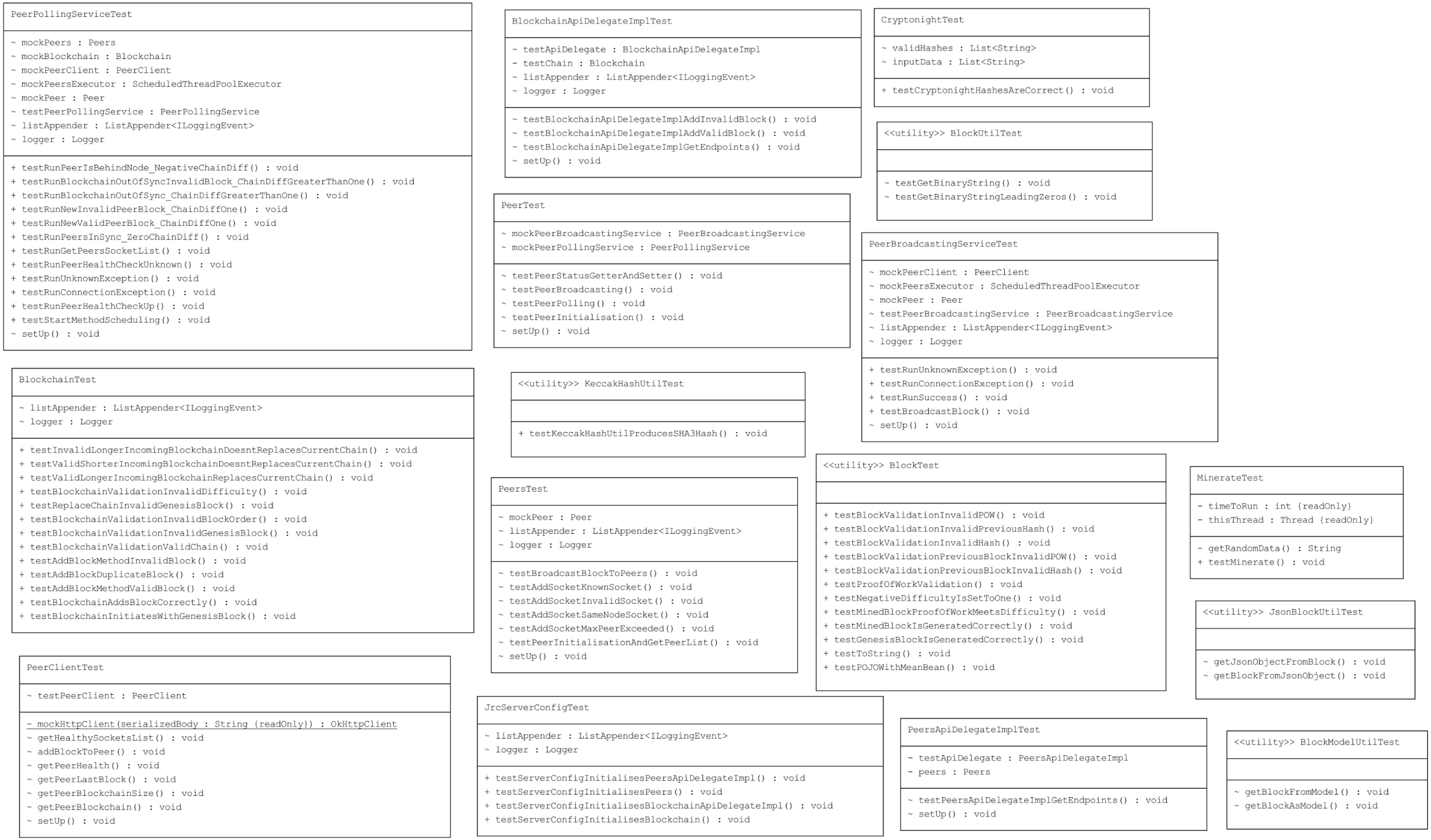
Peer Polling Service Run Method

```java
public class PeerPollingService implements Runnable{
/**
 * The PeerPollingService run task.
 *
 * First the peers health status is checked, and if healthy
 * proceed to get the peers sockets list if it has changed. Then compare
 * the peers blockchain size again this nodes local blockchain instance,
 * and depending on the difference either get the peers last block or
 * attempt to synchronise the whole blockchain if this node is behind the peer
 */
@Override
public void run() {
    log.debug("Attempting to poll peer: [{}]",peer.getPeerSocket());
    try {
        this.peerHealthCheck(); //First check the peers health status
        if(peer.getPeerStatus() == Status.UP){
            // Get the peers healthy peer list
            String peerSocketsList = peerClient.getHealthySocketsList();

            if(!peerSocketsList.equals(cashedPeerSocketsList)){ // The peers peer list has changed, update this nodes peer list...
                log.info("New peers have been detected from the following peer: [{}] !",peer.getPeerSocket());
                peers.addSocketsList(peerSocketsList);
                this.cashedPeerSocketsList = peerSocketsList;
            }

            // Compare the peers blockchain size against this nodes blockchain size...
            int chainSizeDiff = peerClient.getPeerBlockchainSize() - this.blockchain.getChain().size();

            if(chainSizeDiff == 0){ // The peers are in sync as there is no difference in chain size
                log.debug("Peer [{}] is in sync with this node",peer.getPeerSocket());
            }
            if(chainSizeDiff == 1){ // The peers chain is ahead of this node and has the newest block, request and add it...
                log.info("A new block was detected in the following peer: [{}] !",peer.getPeerSocket());
                try {
                    this.blockchain.addBlock(peerClient.getPeerLastBlock());
                } catch (InvalidObjectException e) { // The new block is invalid...
                    log.error("Could not add the new block from peer: [{}]. Reason: {}", peer.getPeerSocket(), e.getMessage());
                }
            }
            if(chainSizeDiff > 1){ // This nodes blockchain is behind the peer by more than one block, attempt to synchronise the blockchain...
                log.info("Attempting to synchronize with the following peer: [{}]", peer.getPeerSocket());
                ArrayList<Block> chainResponse = peerClient.getPeerBlockchain();

                try {
                    Blockchain incomingBlockchain = new Blockchain(chainResponse);
                    this.blockchain.replaceChain(incomingBlockchain);

                    log.info("Successfully synchronized blockchain with the following peer: [{}]!", peer.getPeerSocket());
                } catch (InvalidObjectException e) {
                    // A block in the blockchain is invalid
                    log.error("Could not synchronize with the following peer: [{}]. Reason: {}", peer.getPeerSocket(), e.getMessage());
                }
            }
            if(chainSizeDiff < 0){ // The peers blockchain is behind this nodes
                log.debug("Peer [{}] is behind this node",peer.getPeerSocket());
            }
        }
    } catch (SocketTimeoutException | ConnectException e) { // Currently tested exceptions caused by a lack of connection
        if(peer.getPeerStatus() != Status.DOWN){
            peer.setPeerStatus(Status.DOWN);
            log.info("Could not poll the following peer: [{}]. Reason: {}. Setting peer status to DOWN", peer.getPeerSocket(), e.getMessage());
        }
```

```
63          log.debug("Could not poll the following peer: [{}]. Reason: {}. Peer status is: {}", peer
    .getPeerSocket(), e.getMessage(), peer.getPeerStatus());
64      } catch (Exception e) {
65          peer.setPeerStatus(Status.UNKNOWN);
66          log.debug("Could not poll the following peer: [{}]. Reason: {}. Setting peer status to
    UNKNOWN", peer.getPeerSocket(), e.getMessage());
67          e.printStackTrace();
68      }
69  }
```

Appendix H

Server Application Test Classes' UML Representations


PeerPollingServiceTest
~ mockPeers : Peers
~ mockBlockchain : Blockchain
~ mockPeerClient : PeerClient
~ mockPeersExecutor : ScheduledThreadPoolExecutor
~ mockPeer : Peer
~ testPeerPollingService : PeerPollingService
~ listAppender : ListAppender<ILoggingEvent>
~ logger : Logger
+ testRunPeerIsBehindNode_NegativeChainDiff() : void
+ testRunBlockchainOutOfSyncInvalidBlock_ChainDiffGreaterThanOne() : void
+ testRunBlockchainOutOfSync_ChainDiffGreaterThanOne() : void
+ testRunNewInvalidPeerBlock_ChainDiffOne() : void
+ testRunNewValidPeerBlock_ChainDiffOne() : void
+ testRunPeersInSync_ZeroChainDiff() : void
+ testRunGetPeersSocketList() : void
+ testRunPeerHealthCheckUnknown() : void
+ testRunUnknownException() : void
+ testRunConnectionException() : void
+ testRunPeerHealthCheckUp() : void
+ testStartMethodScheduling() : void
~ setUp() : void
BlockchainApiDelegateImplTest
- testApiDelegate : BlockchainApiDelegateImpl
- testChain : Blockchain
~ listAppender : ListAppender<ILoggingEvent>
~ logger : Logger
~ testBlockchainApiDelegateImplAddInvalidBlock() : void
~ testBlockchainApiDelegateImplAddValidBlock() : void
~ testBlockchainApiDelegateImplGetEndpoints() : void
~ setUp() : void
CryptonightTest
~ validHashes : List<String>
~ inputData : List<String>
+ testCryptonightHashesAreCorrect() : void
<<utility>> BlockUtilTest
~ testGetBinaryString() : void
~ testGetBinaryStringLeadingZeros() : void
PeerTest
~ mockPeerBroadcastingService : PeerBroadcastingService
~ mockPeerPollingService : PeerPollingService
~ testPeerStatusGetterAndSetter() : void
~ testPeerBroadcasting() : void
~ testPeerPolling() : void
~ testPeerInitialisation() : void
~ setUp() : void
PeerBroadcastingServiceTest
~ mockPeerClient : PeerClient
~ mockPeersExecutor : ScheduledThreadPoolExecutor
~ mockPeer : Peer
~ testPeerBroadcastingService : PeerBroadcastingService
~ listAppender : ListAppender<ILoggingEvent>
~ logger : Logger
+ testRunUnknownException() : void
+ testRunConnectionException() : void
+ testRunSuccess() : void
+ testBroadcastBlock() : void
~ setUp() : void
BlockchainTest
~ listAppender : ListAppender<ILoggingEvent>
~ logger : Logger
+ testInvalidLongerIncomingBlockchainDoesntReplacesCurrentChain() : void
+ testValidShorterIncomingBlockchainDoesntReplacesCurrentChain() : void
+ testValidLongerIncomingBlockchainReplacesCurrentChain() : void
+ testBlockchainValidationInvalidDifficulty() : void
+ testReplaceChainInvalidGenesisBlock() : void
+ testBlockchainValidationInvalidBlockOrder() : void
+ testBlockchainValidationInvalidGenesisBlock() : void
+ testBlockchainValidationValidChain() : void
+ testAddBlockMethodInvalidBlock() : void
+ testAddBlockDuplicateBlock() : void
+ testAddBlockMethodValidBlock() : void
+ testBlockchainAddsBlockCorrectly() : void
+ testBlockchainInitiatesWithGenesisBlock() : void
<<utility>> KeccakHashUtilTest
+ testKeccakHashUtilProducesSHA3Hash() : void
<<utility>> BlockTest
+ testBlockValidationInvalidPOW() : void
+ testBlockValidationInvalidPreviousHash() : void
+ testBlockValidationInvalidHash() : void
+ testBlockValidationPreviousBlockInvalidPOW() : void
+ testBlockValidationPreviousBlockInvalidHash() : void
+ testProofOfWorkValidation() : void
+ testNegativeDifficultyIsSetToOne() : void
+ testMinedBlockProofOfWorkMeetsDifficulty() : void
+ testMinedBlockIsGeneratedCorrectly() : void
+ testGenesisBlockIsGeneratedCorrectly() : void
+ testToString() : void
+ testPOJOWithMeanBean() : void
MinerateTest
- timeToRun : int {readOnly}
- thisThread : Thread {readOnly}
- getRandomData() : String
+ testMinerate() : void
PeersTest
~ mockPeer : Peer
~ listAppender : ListAppender<ILoggingEvent>
~ logger : Logger
~ testBroadcastBlockToPeers() : void
~ testAddSocketKnownSocket() : void
~ testAddSocketInvalidSocket() : void
~ testAddSocketSameNodeSocket() : void
~ testAddSocketMaxPeerExceeded() : void
~ testPeerInitialisationAndGetPeerList() : void
~ setUp() : void
<<utility>> JsonBlockUtilTest
~ getJsonObjectFromBlock() : void
~ getBlockFromJsonObject() : void
PeerClientTest
~ testPeerClient : PeerClient
- mockHttpClient(serializedBody : String {readOnly}) : OkHttpClient
~ getHealthySocketsList() : void
~ addBlockToPeer() : void
~ getPeerHealth() : void
~ getPeerLastBlock() : void
~ getPeerBlockchainSize() : void
~ getPeerBlockchain() : void
~ setUp() : void
JrcServerConfigTest
~ listAppender : ListAppender<ILoggingEvent>
~ logger : Logger
+ testServerConfigInitialisesPeersApiDelegateImpl() : void
+ testServerConfigInitialisesPeers() : void
+ testServerConfigInitialisesBlockchainApiDelegateImpl() : void
+ testServerConfigInitialisesBlockchain() : void
PeersApiDelegateImplTest
- testApiDelegate : PeersApiDelegateImpl
- peers : Peers
~ testPeersApiDelegateImplGetEndpoints() : void
~ setUp() : void
<<utility>> BlockModelUtilTest
~ getBlockFromModel() : void
~ getBlockAsModel() : void

Appendix I

Server Application Unit Tests

```
class BlockModelUtilTest {

    @Test
    void getBlockAsModel() {
        //Given
        Block testBlock = new Block("this","is","a","test","block","lol","yeet");

        //When
        BlockModel testBlockModel = BlockModelUtil.getBlockAsModel(testBlock);

        //Then
        assertEquals("this", testBlockModel.getHash());
        assertEquals("is", testBlockModel.getPreviousHash());
        assertEquals("a", testBlockModel.getData());
        assertEquals("test", testBlockModel.getTimeStamp());
        assertEquals("block", testBlockModel.getNonce());
        assertEquals("lol", testBlockModel.getDifficulty());
        assertEquals("yeet", testBlockModel.getProofOfWork());

        assertEquals("class BlockModel {\n" +
                "    hash: this\n" +
                "    previousHash: is\n" +
                "    data: a\n" +
                "    timeStamp: test\n" +
                "    nonce: block\n" +
                "    difficulty: lol\n" +
                "    proofOfWork: yeet\n" +
                "}", testBlockModel.toString());
    }

    @Test
    void getBlockFromModel() {
        //Given
        BlockModel testBlockModel = new BlockModel();

        testBlockModel.setHash("this");
        testBlockModel.setPreviousHash("is");
        testBlockModel.setData("a");
        testBlockModel.setTimeStamp("test");
        testBlockModel.setNonce("block");
        testBlockModel.setDifficulty("model");
        testBlockModel.setProofOfWork("ay");

        //When
        Block testBlock = BlockModelUtil.getBlockFromModel(testBlockModel);

        //Then
        assertEquals("this", testBlock.getHash());
        assertEquals("is", testBlock.getPreviousHash());
        assertEquals("a", testBlock.getData());
        assertEquals("test", testBlock.getTimeStamp());
        assertEquals("block", testBlock.getNonce());
        assertEquals("model", testBlock.getDifficulty());
        assertEquals("ay", testBlock.getProofOfWork());

        assertEquals("Block{hash='this', previousHash='is', data='a', timeStamp='test', nonce='
    block', difficulty='model', proofOfWork='ay'}", testBlock.toString());
    }
}
```

Listing I.1: BlockModelUtilTest test class

```
class PeerModelUtilTest {

    @Test
    void getPeerAsModel() {
        //Given
        //Initialise test peer, executor isn't needed as peer functionality isn't tested in this
    unit test
        Peer testPeer = new Peer("10.10.10.10:8080", null);

        //When
        PeerModel testPeerModel = PeerModelUtil.getPeerAsModel(testPeer);

        //Then
        assertEquals("10.10.10.10:8080", testPeerModel.getPeerSocket());
```

```
        assertEquals("UNKNOWN", testPeerModel.getPeerStatus());
    }
}
```

Listing I.2: PeerModelUtilTest test class

```
class BlockchainApiDelegateImplTest {
    Logger logger = (Logger) LoggerFactory.getLogger(Blockchain.class);
    ListAppender<ILoggingEvent> listAppender = new ListAppender<>();

    private Blockchain testChain;
    private BlockchainApiDelegateImpl testApiDelegate;

    @BeforeEach
    void setUp() {
        //Given

        //Initialise a valid blockchain
        testChain = new Blockchain(new ArrayList<>());
        Block secondBlock = new Block().mineBlock(testChain.getLastBlock(), "Hi, im the second block");
        testChain.getChain().add(secondBlock);
        Block thirdBlock = new Block().mineBlock(testChain.getLastBlock(), "Sup second block im the third block");
        testChain.getChain().add(thirdBlock);
        Block forthBlock = new Block().mineBlock(testChain.getLastBlock(), "And im the fourth block =)");
        testChain.getChain().add(forthBlock);

        //Initialise the api delegate and inject the test blockchain
        testApiDelegate = new BlockchainApiDelegateImpl();
        ReflectionTestUtils.setField(testApiDelegate, "blockchain", testChain);
    }

    @Test
    void testBlockchainApiDelegateImplGetEndpoints() {
        //When
        String blockchainResponse = testApiDelegate.getBlockchain().toString();
        String blockchainSizeResponse = testApiDelegate.getBlockchainSize().toString();
        String lastBlockResponse = testApiDelegate.getLastBlock().toString();

        //Then
        assertEquals("<200 OK OK,[class BlockModel {\n" +
                "    hash: 89b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22\n" +
                "    previousHash: dummyhash\n" +
                "    data: dummydata\n" +
                "    timeStamp: 2020-11-07T19:40:57.585581100Z\n" +
                "    nonce: dummydata\n" +
                "    difficulty: 3\n" +
                "    proofOfWork: 110101110111010001001001100101110101010010000100010111000111110110001101110000100101111110001000000\n" +
                "}, class BlockModel {\n" +
                "    hash: " + testChain.getChain().get(1).getHash() + "\n" +
                "    previousHash: " + testChain.getChain().get(1).getPreviousHash() + "\n" +
                "    data: " + testChain.getChain().get(1).getData() + "\n" +
                "    timeStamp: " + testChain.getChain().get(1).getTimeStamp() + "\n" +
                "    nonce: " + testChain.getChain().get(1).getNonce() + "\n" +
                "    difficulty: " + testChain.getChain().get(1).getDifficulty() + "\n" +
                "    proofOfWork: " + testChain.getChain().get(1).getProofOfWork() + "\n" +
                "}, class BlockModel {\n" +
                "    hash: " + testChain.getChain().get(2).getHash() + "\n" +
                "    previousHash: " + testChain.getChain().get(2).getPreviousHash() + "\n" +
                "    data: " + testChain.getChain().get(2).getData() + "\n" +
                "    timeStamp: " + testChain.getChain().get(2).getTimeStamp() + "\n" +
                "    nonce: " + testChain.getChain().get(2).getNonce() + "\n" +
                "    difficulty: " + testChain.getChain().get(2).getDifficulty() + "\n" +
                "    proofOfWork: " + testChain.getChain().get(2).getProofOfWork() + "\n" +
                "}, class BlockModel {\n" +
                "    hash: " + testChain.getChain().get(3).getHash() + "\n" +
                "    previousHash: " + testChain.getChain().get(3).getPreviousHash() + "\n" +
                "    data: " + testChain.getChain().get(3).getData() + "\n" +
                "    timeStamp: " + testChain.getChain().get(3).getTimeStamp() + "\n" +
                "    nonce: " + testChain.getChain().get(3).getNonce() + "\n" +
                "    difficulty: " + testChain.getChain().get(3).getDifficulty() + "\n" +
                "    proofOfWork: " + testChain.getChain().get(3).getProofOfWork() + "\n" +
```

```
                "}],[]>", blockchainResponse);

        assertEquals("<200 OK OK,class BlockModel {\n" +
                "    hash: " + testChain.getChain().get(3).getHash() + "\n" +
                "    previousHash: " + testChain.getChain().get(3).getPreviousHash() + "\n" +
                "    data: " + testChain.getChain().get(3).getData() + "\n" +
                "    timeStamp: " + testChain.getChain().get(3).getTimeStamp() + "\n" +
                "    nonce: " + testChain.getChain().get(3).getNonce() + "\n" +
                "    difficulty: " + testChain.getChain().get(3).getDifficulty() + "\n" +
                "    proofOfWork: " + testChain.getChain().get(3).getProofOfWork() + "\n" +
                "},[]>", lastBlockResponse);

        assertEquals("<200 OK OK,4,[]>", blockchainSizeResponse);
    }

    @Test //TODO: Also test 200 status successful response code
    void testBlockchainApiDelegateImplAddValidBlock() {
        //Given
        Block newBlock = new Block().mineBlock(testChain.getLastBlock(), "newww blockkkk");
        BlockModel newBlockModel = BlockModelUtil.getBlockAsModel(newBlock);

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        try {
            testApiDelegate.addBlock(newBlockModel);
        } catch (NullPointerException e) { //TODO: Could replace with ReflectionTestUtils, see:
    https://www.baeldung.com/spring-reflection-test-utils
            //Since there is no peers bean, the peers.broadcastBlockToPeers() method call will
    fail,
            //Catch this and fail silently as its not relevant to this unit test and is tested
    during integration testing
        }

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - incorrect logging message displayed","...Block added successfully
    !", logsList.get(2).getMessage());
        assertEquals(newBlock.toString(), testChain.getLastBlock().toString()); //New block is
    the last block in the chain
    }

    @Test
    void testBlockchainApiDelegateImplAddInvalidBlock() {
        //Given
        Block invalidBlock = new Block("I","am","an","evil","block","whos","invalid");
        BlockModel newBlockModel = BlockModelUtil.getBlockAsModel(invalidBlock);

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        String badResponse = testApiDelegate.addBlock(newBlockModel).toString();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("<400 BAD_REQUEST Bad Request,Block validation failed, this block doesn't
    reference the previous blocks hash correctly. Reference to previous hash: am, supplied
    previous blocks hash: " + testChain.getLastBlock().getHash() +  "...,[]>", badResponse);
        assertEquals("failure - incorrect logging message displayed","New incoming block is
    invalid and can't be added to the blockchain. Reason: {}", logsList.get(2).getMessage());
    }
}
```

Listing I.3: BlockchainApiDelegateImplTest test class

```
class PeersApiDelegateImplTest {
    private Peers peers;
    private PeersApiDelegateImpl testApiDelegate;

    @BeforeEach
    void setUp() {
        //Initialise peers with some dummy values
```

```
        peers = new Peers("127.0.0.1:2323", 200, "54.90.44.155:8080,23.444.23.145:2020");

        //Initialise the api delegate and inject the test peers
        testApiDelegate = new PeersApiDelegateImpl();
        ReflectionTestUtils.setField(testApiDelegate, "peers", peers);
    }

    @Test
    void testPeersApiDelegateImplGetEndpoints() {
        //When
        String peersResponse = testApiDelegate.getSocketsList().toString();

        //Then
        assertEquals("<200 OK OK,[class PeerModel {\n" +
                "    peerSocket: 54.90.44.155:8080\n" +
                "    peerStatus: UNKNOWN\n" +
                "}, class PeerModel {\n" +
                "    peerSocket: 23.444.23.145:2020\n" +
                "    peerStatus: UNKNOWN\n" +
                "}],[]>", peersResponse);
    }
}
```

Listing I.4: PeersApiDelegateImplTest test class

```
class BlockUtilTest {

    @Test
    void testGetBinaryStringLeadingZeros() {
        //given
        byte[] in = new byte[] { (byte) 0x00, (byte)0x48, (byte)0x00, (byte)0x03}; //Initialise
    binary array with 9 leading zeros

        int out = BlockUtil.getByteArrayLeadingZeros(in);

        assertEquals("failure - Binary String output is incorrect", 9, out);
    }

    @Test
    void testGetBinaryString() {
        //Given
        String expected = "0000000001001000";
        byte[] in = new byte[] { (byte) 0x00, (byte)0x48}; //Initialise binary array with
    expected binary number

        //When
        String out = BlockUtil.getBinaryString(in);

        //Then
        assertEquals("failure - Binary String has incorrect length", 16, out.length());
        assertEquals("failure - Binary String output is incorrect", expected, out);
    }
}
```

Listing I.5: BlockUtilTest test class

```
class KeccakHashUtilTest {

    @Test //Test that KeccakHashHelper returns a correct SHA3 hash, verified using: https://emn178
    .github.io/online-tools/keccak_256.html
    public void testKeccakHashUtilProducesSHA3Hash(){
        //Given
        String testMessage = "testmessage123";

        //When
        String testDigest = KeccakHashUtil.returnHash(testMessage);

        //Then
        assertEquals("failure - keccak hash helper has returned the wrong hash", "7
    fcae3cd5c246bf2f5dbc8d43d2b2ccc2574472ec87368800a4177e35d57cabe", testDigest);
    }
}
```

Listing I.6: KeccakHashUtilTest test class

```
class BlockchainTest {

    Logger logger = (Logger) LoggerFactory.getLogger(Blockchain.class);
    ListAppender<ILoggingEvent> listAppender = new ListAppender<>();

    @Test
    public void testBlockchainInitiatesWithGenesisBlock(){
        //Given
        Blockchain testChain;
        Block genesisBlock = new Block().genesis();

        //When
        testChain = new Blockchain(new ArrayList<>()); //...a new chain is initialised...

        //Then
        assertEquals("failure - hash of the first block in the blockchain does not equal the genesis hash", genesisBlock.getHash(), testChain.getChain().get(0).getHash());
        assertEquals("failure - data of the first block in the blockchain does not equal the genesis data", genesisBlock.getData(), testChain.getChain().get(0).getData());
        assertEquals("failure - previous hash of the first block in the blockchain does not equal the genesis previous hash", genesisBlock.getPreviousHash(), testChain.getChain().get(0).getPreviousHash());
        assertEquals("failure - timestamp of the first block in the blockchain does not equal the genesis timestamp", genesisBlock.getTimeStamp(), testChain.getChain().get(0).getTimeStamp());
    }

    @Test
    public void testBlockchainAddsBlockCorrectly() throws InvalidObjectException {
        //Given
        Blockchain testChain = new Blockchain(new ArrayList<>());

        //When
        Block secondBlock = new Block().mineBlock(testChain.getLastBlock(), "Hi, im the second block");
        testChain.getChain().add(secondBlock);
        Block thirdBlock = new Block().mineBlock(testChain.getLastBlock(), "Sup second block im the third block");
        testChain.getChain().add(thirdBlock);
        Block forthBlock = new Block().mineBlock(testChain.getLastBlock(), "And im the fourth block =)");
        testChain.getChain().add(forthBlock);

        //Then
        //Check that each block in the chain references the previous block hash correctly
        assertEquals("failure - previous hash of the second block does not reference hash of genesis block", testChain.getChain().get(0).getHash(), testChain.getChain().get(1).getPreviousHash());
        assertEquals("failure - previous hash of the third block does not reference hash of second block", testChain.getChain().get(1).getHash(), testChain.getChain().get(2).getPreviousHash());
        assertEquals("failure - previous hash of the fourth block does not reference hash of third block", testChain.getChain().get(2).getHash(), testChain.getChain().get(3).getPreviousHash());
        //Check that each block has the correct data
        assertEquals("failure - data in the second block is incorrect", "Hi, im the second block", testChain.getChain().get(1).getData());
        assertEquals("failure - data in the second block is incorrect", "Sup second block im the third block", testChain.getChain().get(2).getData());
        assertEquals("failure - data in the second block is incorrect", "And im the fourth block =)", testChain.getChain().get(3).getData());
        //Check that the blockchain has the correct amount of blocks
        assertEquals("failure - blockchain has incorrect amount of blocks", 4, testChain.getChain().size());
    }

    @Test
    public void testAddBlockMethodValidBlock() throws InvalidObjectException {
        //Given
        Blockchain testChain = new Blockchain(new ArrayList<>());
        Block secondBlock = new Block().mineBlock(testChain.getLastBlock(), "Hi, im the second block");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...
```

```
        try {
            testChain.addBlock(secondBlock);
        } catch (NullPointerException e) { //TODO: Could replace with ReflectionTestUtils, see: https://www.baeldung.com/spring-reflection-test-utils
            //Since there is no peers bean, the peers.broadcastBlockToPeers() method call will fail,
            //Catch this and fail silently as its not relevant to this unit test and is tested during integration testing
        }

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - last block in the chain isnt the newly added block...", secondBlock.toString(), testChain.getLastBlock().toString());
        assertEquals("failure - incorrect logging message displayed","...Block added successfully!", logsList.get(2).getMessage());
    }

    @Test
    public void testAddBlockDuplicateBlock() throws InvalidObjectException {
        //Given
        Blockchain testChain = new Blockchain(new ArrayList<>());
        Block secondBlock = new Block().mineBlock(testChain.getLastBlock(), "Hi, im the second block");
        testChain.getChain().add(secondBlock); //Add the second block to the chain directly

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        try {
            testChain.addBlock(secondBlock);
        } catch (NullPointerException e) { //TODO: Could replace with ReflectionTestUtils, see: https://www.baeldung.com/spring-reflection-test-utils
            //Since there is no peers bean, the peers.broadcastBlockToPeers() method call will fail,
            //Catch this and fail silently as its not relevant to this unit test and is tested during integration testing
        }

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - incorrect logging message displayed","Block has already been added to the chain!", logsList.get(1).getMessage());
    }

    @Test
    public void testAddBlockMethodInvalidBlock() {
        //Given
        Blockchain testChain = new Blockchain(new ArrayList<>());
        Block invalidBlock = new Block("I","am","an","evil","block","whos","invalid");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        Exception blockValidationException = null;
        try {
             blockValidationException = assertThrows(InvalidObjectException.class, () -> {
                //Catch the invalid block exception
                testChain.addBlock(invalidBlock);
            });
        } catch (NullPointerException e) { //TODO: Could replace with ReflectionTestUtils, see: https://www.baeldung.com/spring-reflection-test-utils
            //Since there is no peers bean, the peers.broadcastBlockToPeers() method call will fail,
            //Catch this and fail silently as its not relevant to this unit test and is tested during integration testing
        }

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

```

```
        //check exception message
        assertTrue(blockValidationException.getMessage().contains("Block validation failed, this block doesn't reference the previous blocks hash correctly. Reference to previous hash:"));
        //check log message
        assertEquals("failure - incorrect logging message displayed","New incoming block is invalid and can't be added to the blockchain. Reason: {}", logsList.get(2).getMessage());
    }

    @Test
    public void testBlockchainValidationValidChain() throws InvalidObjectException {
        //Given
        Blockchain validChain = new Blockchain(new ArrayList<>());

        //When
        Block secondBlock = new Block().mineBlock(validChain.getLastBlock(), "Hi, im the second block");
        validChain.getChain().add(secondBlock);
        Block thirdBlock = new Block().mineBlock(validChain.getLastBlock(), "Sup second block im the third block");
        validChain.getChain().add(thirdBlock);
        Block forthBlock = new Block().mineBlock(validChain.getLastBlock(), "And im the fourth block =)");
        validChain.getChain().add(forthBlock);

        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        Boolean isChainValid = validChain.isChainValid();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertTrue("failure - valid chain incorrectly flagged as invalid", isChainValid);
        assertEquals("failure - incorrect logging message displayed","Blockchain is valid...", logsList.get(0).getMessage());
    }

    @Test
    public void testBlockchainValidationInvalidGenesisBlock() throws InvalidObjectException {
        //Given
        Block genesisBlock = new Block().genesis();

        List<Block> invalidGenesisChain = new ArrayList<>(); //create a dummy chain as a new arraylist
        Block badBoyBlock = new Block().mineBlock(genesisBlock,"Im a verrrrry bad block whos going to do bad things >:)");
        invalidGenesisChain.add(badBoyBlock); //add a block that isn't the genesis block to the dummy chain

        Blockchain evilBlockchain = new Blockchain(invalidGenesisChain); //Initialise the evil blockchain with the dummy chain

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        boolean isChainValid = evilBlockchain.isChainValid();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertFalse("failure - blockchain with invalid genesis block has verified as valid, big ouf...", isChainValid);
        assertEquals("failure - incorrect logging message displayed","Chain is invalid, first block in the chain is not genesis block...", logsList.get(0).getMessage());
    }

    @Test
    public void testBlockchainValidationInvalidBlockOrder(){
        //Given
        List<Block> invalidPreviousHashChain = new ArrayList<>(); //create a dummy chain as a new arraylist

        invalidPreviousHashChain.add(new Block().genesis()); //start the chain with a valid genesis block
        invalidPreviousHashChain.add(new Block().mineBlock(new Block().genesis(), "a valid second
```

```
 block")); //add a valid block after the genesis block

        Block thirdBlock = new Block().mineBlock(new Block().genesis(),"Im meant to be the second block in the chain");
        Block forthBlock = new Block().mineBlock(thirdBlock,"Im meant to be the second block in the chain");

        invalidPreviousHashChain.add(forthBlock);
        invalidPreviousHashChain.add(thirdBlock); //switch around the second and third blocks in the chain

        Blockchain evilBlockchain = new Blockchain(invalidPreviousHashChain); //Initialise the evil blockchain with the dummy chain

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        Exception blockValidationException = assertThrows(InvalidObjectException.class, () -> {
            //Catch the invalid block exception
            evilBlockchain.isChainValid();
        });

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //check exception message
        assertTrue(blockValidationException.getMessage().contains("Block validation failed, this block doesn't reference the previous blocks hash correctly. Reference to previous hash:"));
        //check log message
        assertEquals("Chain is invalid, the block {} in the chain is invalid.", logsList.get(0).getMessage());
    }

    @Test
    public void testReplaceChainInvalidGenesisBlock() throws InvalidObjectException {
        //Given
        Blockchain testChain = new Blockchain(new ArrayList<>());

        Block genesisBlock = new Block().genesis();

        List<Block> invalidGenesisChain = new ArrayList<>(); //create a dummy chain as a new arraylist
        Block badBoyBlock = new Block().mineBlock(genesisBlock,"Im a verrrrry bad block whos going to do bad things >:)");
        invalidGenesisChain.add(badBoyBlock); //add a block that isn't the genesis block to the dummy chain

        Blockchain evilBlockchain = new Blockchain(invalidGenesisChain); //Initialise the evil blockchain with the dummy chain
        evilBlockchain.getChain().add(genesisBlock); //Add the genesis block as second block in the evil chain

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testChain.replaceChain(evilBlockchain);

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - incorrect logging message displayed","Chain is invalid, first block in the chain is not genesis block...", logsList.get(1).getMessage());
    }

    @Test
    public void testBlockchainValidationInvalidDifficulty() throws InvalidObjectException {
        //Given
        List<Block> invalidPreviousHashChain = new ArrayList<>(); //create a dummy chain as a new arraylist

        Block genesisBlock = new Block().genesis();

        //Generate a block based on the previously generated valid second block, but set jump the difficulty more than 1
```

```
        Block evilSecondBlock = new Block("
    fb6bf0dd384664830a642467b184c2473fa6b21251d9c45b333e9f55505294c7", "89
    b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22", "secondBlockData", "
    2020-11-14T23:55:10.208261300Z", "4", "69", "
    1111110001001111000001100000100101100111011000111000110111111010011000101010001111110100101001011001
    ");

        invalidPreviousHashChain.add(genesisBlock);
        invalidPreviousHashChain.add(evilSecondBlock); //Add the invalid block to the dummy chain

        Blockchain evilBlockchain = new Blockchain(invalidPreviousHashChain); //Initialise the
    evil blockchain with the dummy chain

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        Boolean isChainValid = evilBlockchain.isChainValid();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertFalse("failure - blockchain that has a difficulty jump has verified as valid, big
    ouf...", isChainValid);
        assertEquals("failure - incorrect logging message displayed","Chain is invalid, the block
     {} in the chain has a difficulty jump greater than 1. Difficulty changed by: {}...", logsList
    .get(0).getMessage());
    }

    @Test
    public void testValidLongerIncomingBlockchainReplacesCurrentChain() throws
    InvalidObjectException {
        //Given
        Blockchain smolChain = new Blockchain(new ArrayList<>());
        Blockchain bigChain = new Blockchain(new ArrayList<>());

        Block smolChainBlock = new Block().mineBlock(smolChain.getLastBlock(), "im the only block
     in this chain =(");
        smolChain.getChain().add(smolChainBlock);

        Block secondBlock = new Block().mineBlock(bigChain.getLastBlock(), "Hi, im the second
    block");
        bigChain.getChain().add(secondBlock);
        Block thirdBlock = new Block().mineBlock(bigChain.getLastBlock(), "Sup second block im
    the third block");
        bigChain.getChain().add(thirdBlock);
        Block forthBlock = new Block().mineBlock(bigChain.getLastBlock(), "And im the fourth
    block =)");
        bigChain.getChain().add(forthBlock);

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        smolChain.replaceChain(bigChain);

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - Original blockchains chain was not replaced by the longer
    incoming chain", bigChain.getChain(), smolChain.getChain());
        assertEquals("failure - Original blockchains chain is incorrect length after replacement"
    , 4, smolChain.getChain().size());

        assertEquals("failure - incorrect logging message displayed","Blockchain is valid...",
    logsList.get(1).getMessage());
        assertEquals("failure - incorrect logging message displayed","Chain replacement
    successful...", logsList.get(2).getMessage());
    }

    @Test
    public void testValidShorterIncomingBlockchainDoesntReplacesCurrentChain() throws
    InvalidObjectException {
        //Given
        Blockchain smolChain = new Blockchain(new ArrayList<>());
        Blockchain bigChain = new Blockchain(new ArrayList<>());
```

```
300
301         Block smolChainBlock = new Block().mineBlock(smolChain.getLastBlock(), "im the only block
     in this chain =(");
302         smolChain.getChain().add(smolChainBlock);
303
304         Block secondBlock = new Block().mineBlock(bigChain.getLastBlock(), "Hi, im the second
    block");
305         bigChain.getChain().add(secondBlock);
306         Block thirdBlock = new Block().mineBlock(bigChain.getLastBlock(), "Sup second block im
    the third block");
307         bigChain.getChain().add(thirdBlock);
308         Block forthBlock = new Block().mineBlock(bigChain.getLastBlock(), "And im the fourth
    block =)");
309         bigChain.getChain().add(forthBlock);
310
311         //When
312         listAppender.start();
313         logger.addAppender(listAppender); //start log capture...
314
315         bigChain.replaceChain(smolChain);
316
317         //Then
318         List<ILoggingEvent> logsList = listAppender.list; //...store captured logs
319
320         assertNotEquals("failure - Original blockchains chain was replaced by smaller chain, big
    ouf", bigChain.getChain(), smolChain.getChain());
321         assertEquals("failure - incorrect logging message displayed","Incoming blockchain is not
    longer than current blockchain...", logsList.get(1).getMessage());
322     }
323
324     @Test
325     public void testInvalidLongerIncomingBlockchainDoesntReplacesCurrentChain(){
326         //Given
327         Blockchain smolChain = new Blockchain(new ArrayList<>());
328         Block smolChainBlock = new Block().mineBlock(smolChain.getLastBlock(), "im the only block
     in this chain =(");
329         smolChain.getChain().add(smolChainBlock);
330
331         List<Block> invalidPreviousHashChain = new ArrayList<>(); //create a dummy chain as a new
     arraylist
332
333         invalidPreviousHashChain.add(new Block().genesis()); //start the chain with a valid
    genesis block
334         invalidPreviousHashChain.add(new Block().mineBlock(new Block().genesis(), "a valid second
     block")); //add a valid block after the genesis block
335
336         Block thirdBlock = new Block().mineBlock(new Block().genesis(),"Im meant to be the second
     block in the chain");
337         Block forthBlock = new Block().mineBlock(thirdBlock,"Im meant to be the second block in
    the chain");
338
339         invalidPreviousHashChain.add(forthBlock);
340         invalidPreviousHashChain.add(thirdBlock); //switch around the second and third blocks in
    the chain
341
342         Blockchain evilBigBlockchain = new Blockchain(invalidPreviousHashChain); //Initialise the
     evil blockchain with the dummy chain
343
344         //When
345         listAppender.start();
346         logger.addAppender(listAppender); //start log capture...
347
348         Exception blockValidationException = assertThrows(InvalidObjectException.class, () -> {
349             //Catch the invalid block exception
350             smolChain.replaceChain(evilBigBlockchain);;
351         });
352
353         //Then
354         List<ILoggingEvent> logsList = listAppender.list; //...store captured logs
355
356         assertNotEquals("failure - Original blockchains chain was replaced by invalid chain, big
    ouf", evilBigBlockchain.getChain(), smolChain.getChain());
357         assertEquals("failure - Original blockchains chain is incorrect length", 2, smolChain.
    getChain().size());
358         //check exception message
359         assertTrue(blockValidationException.getMessage().contains("Block validation failed, this
```

```
    block doesn't reference the previous blocks hash correctly. Reference to previous hash:"));
360         //check log messages
361         assertEquals("Chain is invalid, the block {} in the chain is invalid.", logsList.get(1).
    getMessage());
362     }
363 }
```

Listing I.7: BlockchainTest test class

```
1  class BlockTest {
2
3      @Test //MeanBean POJO tester will test getters/setters
4      public void testPOJOWithMeanBean(){
5          BeanTester tester = new BeanTester();
6          tester.testBean(Block.class);
7      }
8
9      @Test //Test ensures any new variables added to Block will be added to toString method
10     public void testToString(){
11         ToStringVerifier.forClass(Block.class).withClassName(NameStyle.SIMPLE_NAME).verify();
12     }
13
14     @Test //Test that a block has all the correct genesis data when .genesis() method is called
15     public void testGenesisBlockIsGeneratedCorrectly(){
16         //Given
17         String expectedPreviousHash = "dummyhash";
18         String expectedData = "dummydata";
19         String expectedTimeStamp = "2020-11-07T19:40:57.585581100Z";
20         String expectedNonce = "dummydata";
21         String expectedDifficulty = "3";
22         String expectedProofOfWork = "
    1101011101110100010010011001011010101001000010001011100011111011000110111000010010111111000100000
    ";
23
24         String expectedHash = "89b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22";
25
26         Block genesisBlock = new Block();
27
28         //When
29         genesisBlock.genesis();
30
31         //Then
32         assertEquals("failure - previousHash of the genesis block is incorrect",
    expectedPreviousHash, genesisBlock.getPreviousHash());
33         assertEquals("failure - data of the genesis block is incorrect", expectedData,
    genesisBlock.getData());
34         assertEquals("failure - timeStamp of the genesis block is incorrect", expectedTimeStamp,
    genesisBlock.getTimeStamp());
35         assertEquals("failure - nonce of the genesis block is incorrect", expectedNonce,
    genesisBlock.getNonce());
36         assertEquals("failure - difficulty of the genesis block is incorrect", expectedDifficulty
    , genesisBlock.getDifficulty());
37         assertEquals("failure - proofOfWork of the genesis block is incorrect",
    expectedProofOfWork, genesisBlock.getProofOfWork());
38
39         assertEquals("failure - generated genesis block hash is incorrect", expectedHash,
    genesisBlock.getHash());
40     }
41
42     @Test //Test that .mineBlock() correctly sets previousHash value of a newly mined block to the
     hash value of the previous block, and that the block data is correct
43     public void testMinedBlockIsGeneratedCorrectly(){
44         //Given
45         String genesisHash = "89b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22";
46
47         Block genesisBlock = new Block();
48         genesisBlock.genesis();
49
50         Block secondBlock = new Block();
51         Block thirdBlock = new Block();
52
53         //When
54         secondBlock.mineBlock(genesisBlock,"secondBlockData");
55         thirdBlock.mineBlock(secondBlock,"thirdBlockData");
56
57         //Then (note: cannot assert generated hash of second block as this will always be
```

```
    different due to changing timestamp)
58      assertEquals("failure - previousHash of second block does not equal genesis hash",
    genesisHash, secondBlock.getPreviousHash());
59      assertEquals("failure - data of second block is incorrect", "secondBlockData",
    secondBlock.getData());
60
61      assertEquals("failure - previousHash of third block does not equal second block hash",
    secondBlock.getHash(), thirdBlock.getPreviousHash());
62      assertEquals("failure - previousHash of third block does not equal second block hash", "
    thirdBlockData", thirdBlock.getData());
63      assertNotEquals("failure - hash of second block is the same as third block",secondBlock.
    getHash(),thirdBlock.getHash());
64  }
65
66  @Test //Test difficulty is adjusted during .mineBlock() method
67  public void testMinedBlockProofOfWorkMeetsDifficulty(){
68      //Given
69      Block genesisBlock = new Block();
70      genesisBlock.genesis();
71
72      Block secondBlock = new Block();
73      Block thirdBlock = new Block();
74
75      //When
76      secondBlock.mineBlock(genesisBlock,"Difficulty should reduce");
77      thirdBlock.mineBlock(secondBlock,"Difficulty should increase");
78
79      long secondBlockDiff = BlockUtil.calcBlockTimeDiff(secondBlock.getTimeStamp(),
    genesisBlock.getTimeStamp()); //Time taken between genesis block was mined, and second block
    was mined
80      long thirdBlockDiff = BlockUtil.calcBlockTimeDiff(thirdBlock.getTimeStamp(),secondBlock.
    getTimeStamp()); //Time taken between second block was mined, and third block was mined
81
82      //Then
83      assertEquals("failure - difficulty of the genesis block is incorrect", "3", genesisBlock.
    getDifficulty());
84
85      assertThat("Time taken between genesis block and second block should be greater than 120
    seconds", secondBlockDiff, greaterThan(120L));
86      assertEquals(String.format("failure - Difference between second block and previous block
    is: %d , therefore difficulty should have decreased", secondBlockDiff) , "2", secondBlock.
    getDifficulty());
87
88      //Below assert can fail correctly, however this is incredibly unlikely unless (as noted
    in failure reason) the cpu that this is ran on is unbelievably slow
89      assertThat("Time taken between genesis block and third block should be less than 120
    seconds, although you might just have a reallllllllly bad cpu =p", thirdBlockDiff, lessThan(120
    L));
90      assertEquals(String.format("failure - Difference between third block and previous block
    is: %d , therefore difficulty should have increased", thirdBlockDiff) , "3", thirdBlock.
    getDifficulty());
91  }
92
93  @Test //Test that if the previous difficulty is negative, the next block difficulty is set to
    one
94  public void testNegativeDifficultyIsSetToOne(){
95      //Given
96      Block negativeDifficultyBlock = new Block("test", "test", "test", "2020-11-07T19
    :40:57.585581100Z", "69", "-5", "test");
97      Block nextBlock = new Block();
98
99      //When
100     nextBlock.mineBlock(negativeDifficultyBlock, "difficulty should be set to one");
101
102     //Then
103     assertEquals("failure - difficulty of the genesis block is incorrect", "1", nextBlock.
    getDifficulty());
104 }
105
106 @Test //Test the proof of work validation method correctly verifies POW strings
107 public void testProofOfWorkValidation(){
108     //Given
109     Block validProofOfWorkBlock = new Block("test", "test", "test", "2020-11-07T19
    :40:57.585581100Z", "69", "3", "
    11011010010001100110011110110001110111001101100100001001011100001101000000000110100001011110010010
    ");
```

```
        Block invalidProofOfWorkBlock = new Block("test", "test", "test", "2020-11-07T19
    :40:57.585581100Z", "69", "3", "invalid");

        //When
        Boolean shouldBeValid = validProofOfWorkBlock.isProofOfWorkValid();
        Boolean shouldBeInvalid = invalidProofOfWorkBlock.isProofOfWorkValid();

        //Then
        Assert.assertTrue("Valid proof of work is being verified as invalid", shouldBeValid);
        Assert.assertFalse("Invalid proof of work is being verified as valid", shouldBeInvalid);
    }

    @Test
    public void testBlockValidationPreviousBlockInvalidHash(){
        //Given
        Block prevGenesisBlock = new Block().genesis();
        Block invalidBlock = new Block("I","am","an","evil","block","whos","invalid");

        //When
        Exception blockValidationException = assertThrows(InvalidObjectException.class, () -> {
            //Catch the invalid block exception
            prevGenesisBlock.validateBlock(invalidBlock);
        });

        //Then
        assertTrue(blockValidationException.getMessage().contains("Block validation failed,
    supplied previous block has an invalid hash. Supplied previous block hash:"));
    }

    @Test
    public void testBlockValidationPreviousBlockInvalidPOW(){
        //Given
        Block prevGenesisBlock = new Block().genesis();
        Block invalidProofOfWorkBlock = new Block("48
    bb2616fe1332d9113a2a4c2813e5db7b37613093e8aabfe4261b44ed52a963", "89
    b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22", "test", "2020-11-07T19
    :40:57.585581100Z", "69", "3", "invalid");

        //When
        Exception blockValidationException = assertThrows(InvalidObjectException.class, () -> {
            //Catch the invalid block exception
            prevGenesisBlock.validateBlock(invalidProofOfWorkBlock);
        });

        //Then
        assertTrue(blockValidationException.getMessage().contains("Block validation failed,
    supplied previous block has an invalid proof of work..."));
    }

    @Test
    public void testBlockValidationInvalidHash(){
        //Given
        Block prevGenesisBlock = new Block().genesis();
        Block invalidBlock = new Block("I","89
    b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22","an","evil","block","whos","
    invalid");

        //When
        Exception blockValidationException = assertThrows(InvalidObjectException.class, () -> {
            //Catch the invalid block exception
            invalidBlock.validateBlock(prevGenesisBlock);
        });

        //Then
        assertTrue(blockValidationException.getMessage().contains("Block validation failed, this
    block has an incorrect hash value. This blocks hash:"));
    }

    @Test
    public void testBlockValidationInvalidPreviousHash(){
        //Given
        Block prevGenesisBlock = new Block().genesis();
        Block invalidBlock = new Block("I","am","an","evil","block","whos","invalid");

        //When
        Exception blockValidationException = assertThrows(InvalidObjectException.class, () -> {
```

```
            //Catch the invalid block exception
            invalidBlock.validateBlock(prevGenesisBlock);
        });

        //Then
        assertTrue(blockValidationException.getMessage().contains("Block validation failed, this block doesn't reference the previous blocks hash correctly. Reference to previous hash:"));
    }

    @Test
    public void testBlockValidationInvalidPOW(){
        //Given
        Block prevGenesisBlock = new Block().genesis();
        Block invalidProofOfWorkBlock = new Block("48bb2616fe1332d9113a2a4c2813e5db7b37613093e8aabfe4261b44ed52a963", "89b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22", "test", "2020-11-07T19:40:57.585581100Z", "69", "3", "invalid");

        //When
        Exception blockValidationException = assertThrows(InvalidObjectException.class, () -> {
            //Catch the invalid block exception
            invalidProofOfWorkBlock.validateBlock(prevGenesisBlock);
        });

        //Then
        assertTrue(blockValidationException.getMessage().contains("Block validation failed, this block has an incorrect proof of work..."));
    }
}
```

Listing I.8: BlockTest test class

```
class CryptonightTest {
    List<String> inputData = Arrays
            .asList("This is a test This is a test This is a test",
                    "Lorem ipsum dolor sit amet, consectetur adipiscing",
                    "elit, sed do eiusmod tempor incididunt ut labore",
                    "et dolore magna aliqua. Ut enim ad minim veniam,",
                    "quis nostrud exercitation ullamco laboris nisi",
                    "ut aliquip ex ea commodo consequat. Duis aute",
                    "irure dolor in reprehenderit in voluptate velit",
                    "esse cillum dolore eu fugiat nulla pariatur.",
                    "Excepteur sint occaecat cupidatat non proident,",
                    "sunt in culpa qui officia deserunt mollit anim id est laborum.",
                    //Above 10 hash tests are taken from Monero CryptoNightV2 implementation (which the CryptoNightJNI component is based off: https://github.com/monero-project/monero/commit/f3cd51a12b202875bd8191668aceb8a4f810ecd4)
                    "This is a test",
                    "",
                    "oioi im brit-ish",
                    "abc",
                    "x123!$$ *^(%)$$$$");

    List<String> validHashes = Arrays
            .asList("353fdc068fd47b03c04b9431e005e00b68c2168a3cc7335c8b9b308156591a4f",
                    "72f134fc50880c330fe65a2cb7896d59b2e708a0221c6a9da3f69b3a702d8682",
                    "410919660ec540fc49d8695ff01f974226a2a28dbbac82949c12f541b9a62d2f",
                    "4472fecfeb371e8b7942ce0378c0ba5e6d0c6361b669c587807365c787ae652d",
                    "577568395203f1f1225f2982b637f7d5e61b47a0f546ba16d46020b471b74076",
                    "f6fd7efe95a5c6c4bb46d9b429e3faf65b1ce439e116742d42b928e61de52385",
                    "422f8cfe8060cf6c3d9fd66f68e3c9977adb683aea2788029308bbe9bc50d728",
                    "512e62c8c8c833cfbd9d361442cb00d63c0a3fd8964cfd2fedc17c7c25ec2d4b",
                    "12a794c1aa13d561c9c6111cee631ca9d0a321718d67d3416add9de1693ba41e",
                    "2659ff95fc74b6215c1dc741e85b7a9710101b30620212f80eb59c3c55993f9d",
                    //Above 10 hash tests are taken from Monero CryptoNightV2 implementation (which the CryptoNightJNI component is based off: https://github.com/monero-project/monero/commit/f3cd51a12b202875bd8191668aceb8a4f810ecd4)
                    "c1d6521259b6a9d29eb19df3895c601bcb9ae1811ec3dd175a4a9c2949af14fe",
                    "e34985722288be50a2068f973f02248d62e7bc6a0a0dfca2eb84909724857a72",
                    "72bf63a5503919c9d25b9eaaaa50eac6fc1a93e852c4a1fced9b8246da2a22ab",
                    "15b5d19b1a77580c49be0560154d94aace754e6640388e2d738a0ab77f3f2c07",
                    "b29d5abcb136383eefef46d8f1142f8f7787156f15ae6823c5c9d5eef19cce35");

    @Test //Test duplicated in use Cryptonight component: https://github.com/jounaidr/CryptoNightJNI
    public void testCryptonightHashesAreCorrect(){
```

```
        Cryptonight cryptonight;
        String out;

        //Inorder for test to run in CI, each hash check must be done sequentially (not in for loop)
        cryptonight = new Cryptonight(inputData.get(0));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(0)),validHashes.get(0),out);

        cryptonight = new Cryptonight(inputData.get(1));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(1)),validHashes.get(1),out);

        cryptonight = new Cryptonight(inputData.get(2));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(2)),validHashes.get(2),out);

        cryptonight = new Cryptonight(inputData.get(3));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(3)),validHashes.get(3),out);

        cryptonight = new Cryptonight(inputData.get(4));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(4)),validHashes.get(4),out);

        cryptonight = new Cryptonight(inputData.get(5));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(5)),validHashes.get(5),out);

        cryptonight = new Cryptonight(inputData.get(6));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(6)),validHashes.get(6),out);

        cryptonight = new Cryptonight(inputData.get(7));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(7)),validHashes.get(7),out);

        cryptonight = new Cryptonight(inputData.get(8));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(8)),validHashes.get(8),out);

        cryptonight = new Cryptonight(inputData.get(9));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(9)),validHashes.get(9),out);

        cryptonight = new Cryptonight(inputData.get(10));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(10)),validHashes.get(10),out);

        cryptonight = new Cryptonight(inputData.get(11));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(11)),validHashes.get(11),out);

        cryptonight = new Cryptonight(inputData.get(12));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(12)),validHashes.get(12),out);

        cryptonight = new Cryptonight(inputData.get(13));
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData.get(13)),validHashes.get(13),out);

        cryptonight = new Cryptonight(inputData.get(14));
```

```
        out = new String(Hex.encode(cryptonight.returnHash()));
        assertEquals(String.format("The following message was incorrectly hashed: %s ", inputData
    .get(14)),validHashes.get(14),out);
    }
}
```

Listing I.9: CryptonightTest test class

```
class PeerBroadcastingServiceTest {
    Logger logger = (Logger) LoggerFactory.getLogger(PeerBroadcastingService.class);
    ListAppender<ILoggingEvent> listAppender = new ListAppender<>();

    PeerBroadcastingService testPeerBroadcastingService;

    @Mock
    Peer mockPeer;

    @Mock
    ScheduledThreadPoolExecutor mockPeersExecutor;

    @Mock
    PeerClient mockPeerClient;

    @BeforeEach
    void setUp() {
        //Given
        MockitoAnnotations.initMocks(this);
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);

        testPeerBroadcastingService = new PeerBroadcastingService(mockPeer, mockPeersExecutor);
        ReflectionTestUtils.setField(testPeerBroadcastingService, "peerClient", mockPeerClient);
    }

    @Test
    public void testBroadcastBlock(){
        //Given
        Block testBlock = new Block("this","is","a","test","block","lol","yeet");

        //When
        testPeerBroadcastingService.broadcastBlock(testBlock);

        //Then
        //Check blockToBroadcast is set correctly
        assertEquals(testBlock.toString(), ReflectionTestUtils.getField(
    testPeerBroadcastingService, "blockToBroadcast").toString());
        //Verify executor submits the runnable task once
        Mockito.verify(mockPeersExecutor, times(1)).submit(testPeerBroadcastingService);
    }

    @Test
    public void testRunSuccess() throws IOException, JSONException {
        //Given
        Block testBlock = new Block("this","is","a","test","block","lol","yeet");;
        ReflectionTestUtils.setField(testPeerBroadcastingService, "blockToBroadcast", testBlock);

        //When
        testPeerBroadcastingService.run();

        //Then
        //Verify the peer client addBlockToPeer method is called with the testBlock
        Mockito.verify(mockPeerClient, times(1)).addBlockToPeer(testBlock);
    }

    @Test
    public void testRunConnectionException() throws IOException, JSONException {
        //Given
        Block testBlock = new Block("this","is","a","test","block","lol","yeet");;
        ReflectionTestUtils.setField(testPeerBroadcastingService, "blockToBroadcast", testBlock);

        Mockito.when(mockPeerClient.addBlockToPeer(testBlock)).thenThrow(new ConnectException());

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerBroadcastingService.run();
```

```
68
69          //Then
70          List<ILoggingEvent> logsList = listAppender.list; //...store captured logs
71
72          //Verify status is set to DOWN
73          Mockito.verify(mockPeer, times(1)).setPeerStatus(Status.DOWN);
74          //Verify log message
75          assertEquals("failure - incorrect logging message displayed","Could not broadcast to the
    following peer: [{}]. Reason: {}. Setting peer status to DOWN", logsList.get(1).getMessage());
76      }
77
78      @Test
79      public void testRunUnknownException() throws IOException, JSONException {
80          //Given
81          Block testBlock = new Block("this","is","a","test","block","lol","yeet");;
82          ReflectionTestUtils.setField(testPeerBroadcastingService, "blockToBroadcast", testBlock);
83
84          Mockito.when(mockPeerClient.addBlockToPeer(testBlock)).thenThrow(new NullPointerException
    ());
85
86          //When
87          listAppender.start();
88          logger.addAppender(listAppender); //start log capture...
89
90          testPeerBroadcastingService.run();
91
92          //Then
93          List<ILoggingEvent> logsList = listAppender.list; //...store captured logs
94
95          //Verify status is set to DOWN
96          Mockito.verify(mockPeer, times(1)).setPeerStatus(Status.UNKNOWN);
97          //Verify log message
98          assertEquals("failure - incorrect logging message displayed","Could not broadcast to the
    following peer: [{}]. Reason: {}. Setting peer status to UNKNOWN", logsList.get(1).getMessage
    ());
99      }
100 }
```

Listing I.10: PeerBroadcastingServiceTest test class

```
1  class PeerPollingServiceTest {
2      Logger logger = (Logger) LoggerFactory.getLogger(PeerPollingService.class);
3      ListAppender<ILoggingEvent> listAppender = new ListAppender<>();
4
5      PeerPollingService testPeerPollingService;
6
7      @Mock
8      Peer mockPeer;
9
10     @Mock
11     ScheduledThreadPoolExecutor mockPeersExecutor;
12
13     @Mock
14     PeerClient mockPeerClient;
15
16     @Mock
17     Blockchain mockBlockchain;
18
19     @Mock
20     Peers mockPeers;
21
22     @BeforeEach
23     void setUp() {
24         //Given
25         MockitoAnnotations.initMocks(this);
26
27         testPeerPollingService = new PeerPollingService(mockPeer, mockPeersExecutor);
28         ReflectionTestUtils.setField(testPeerPollingService, "peerClient", mockPeerClient);
29         ReflectionTestUtils.setField(testPeerPollingService, "blockchain", mockBlockchain);
30         ReflectionTestUtils.setField(testPeerPollingService, "peers", mockPeers);
31     }
32
33     @Test
34     public void testStartMethodScheduling(){
35         //When
36         testPeerPollingService.start();
```

```
        //Then
        //Verify executor schedules the task correctly
        Mockito.verify(mockPeersExecutor, times(1)).scheduleAtFixedRate(
                any(PeerPollingService.class),
                any(long.class), //This covers the random delay method as well
                any(long.class),
                any(TimeUnit.class));
    }

    @Test
    public void testRunPeerHealthCheckUp() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeerClient.getPeerHealth()).thenReturn("UP");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Check the peer status is set to UP once
        Mockito.verify(mockPeer, times(1)).setPeerStatus(Status.UP);
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Connection reestablished
    with peer: [{}] ! Setting peer status to UP", logsList.get(1).getMessage());
    }

    @Test
    public void testRunConnectionException() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeerClient.getPeerHealth()).thenThrow(new ConnectException());

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Check the peer status is set to DOWN on connection exception
        Mockito.verify(mockPeer, times(1)).setPeerStatus(Status.DOWN);
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Could not poll the
    following peer: [{}]. Reason: {}. Setting peer status to DOWN", logsList.get(1).getMessage());
    }

    @Test
    public void testRunUnknownException() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeerClient.getPeerHealth()).thenThrow(new NullPointerException()); //
    Throw an 'unknown' exception

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Check the peer status is set to UNKNOWN
        Mockito.verify(mockPeer, times(1)).setPeerStatus(Status.UNKNOWN);
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Could not poll the
    following peer: [{}]. Reason: {}. Setting peer status to UNKNOWN", logsList.get(1).getMessage
    ());
    }

    @Test
```

```
    public void testRunPeerHealthCheckUnknown() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeerClient.getPeerHealth()).thenReturn("this is an invalid satus response");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Check the peer status is set to UP once
        Mockito.verify(mockPeer, times(1)).setPeerStatus(Status.UNKNOWN);
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Peer health check returned invalid response: {}. Setting peer [{}] status to UNKNOWN", logsList.get(1).getMessage());
    }

    @Test
    public void testRunGetPeersSocketList() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);
        Mockito.when(mockPeerClient.getHealthySocketsList()).thenReturn("10.10.10.10:8080");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Verify the new socketlist is added to peers
        Mockito.verify(mockPeers, times(1)).addSocketsList("10.10.10.10:8080");
        //Verify the cashed sockets list is update
        assertEquals("10.10.10.10:8080", ReflectionTestUtils.getField(testPeerPollingService, "cashedPeerSocketsList"));
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","New peers have been detected from the following peer: [{}] !", logsList.get(1).getMessage());
    }

    @Test
    public void testRunPeersInSync_ZeroChainDiff() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);

        Mockito.when(mockPeerClient.getHealthySocketsList()).thenReturn("");
        Mockito.when(mockPeerClient.getPeerBlockchainSize()).thenReturn(0); //will result in 0 diff

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Peer [{}] is in sync with this node", logsList.get(2).getMessage());
    }

    @Test
    public void testRunNewValidPeerBlock_ChainDiffOne() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);

        Mockito.when(mockPeerClient.getHealthySocketsList()).thenReturn("");
        Mockito.when(mockPeerClient.getPeerBlockchainSize()).thenReturn(1); //Will result in a
```

```
    diff on 1

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Verify the new block is added to the blockchain
        Mockito.verify(mockBlockchain, times(1)).addBlock(any());
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","A new block was detected in
     the following peer: [{}] !", logsList.get(2).getMessage());
    }

    @Test
    public void testRunNewInvalidPeerBlock_ChainDiffOne() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);

        Mockito.when(mockPeerClient.getHealthySocketsList()).thenReturn("");
        Mockito.when(mockPeerClient.getPeerBlockchainSize()).thenReturn(1); //Will result in a
    diff on 1
        Mockito.when(mockPeerClient.getPeerLastBlock()).thenThrow(new InvalidObjectException("
    test exception"));

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Could not add the new block
     from peer: [{}]. Reason: {}", logsList.get(3).getMessage());
    }

    @Test
    public void testRunBlockchainOutOfSync_ChainDiffGreaterThanOne() throws IOException,
    JSONException {
        //Given
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);

        Mockito.when(mockPeerClient.getHealthySocketsList()).thenReturn("");
        Mockito.when(mockPeerClient.getPeerBlockchainSize()).thenReturn(3); //Will result in a
    diff greater than 1

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Verify peer blockchain call to client, and replace of the node blockchain...
        Mockito.verify(mockPeerClient, times(1)).getPeerBlockchain();
        Mockito.verify(mockBlockchain, times(1)).replaceChain(any());
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Attempting to synchronize
    with the following peer: [{}]", logsList.get(2).getMessage());
        assertEquals("failure - incorrect logging message displayed","Successfully synchronized
    blockchain with the following peer: [{}] !", logsList.get(3).getMessage());
    }

    @Test
    public void testRunBlockchainOutOfSyncInvalidBlock_ChainDiffGreaterThanOne() throws
    IOException, JSONException {
        //Given
```

```
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);

        Mockito.when(mockPeerClient.getHealthySocketsList()).thenReturn("");
        Mockito.when(mockPeerClient.getPeerBlockchainSize()).thenReturn(3); //Will result in a diff greater than 1

        Mockito.doThrow(new InvalidObjectException("test exception")).when(mockBlockchain).replaceChain(any());

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Verify peer blockchain call to client, and replace of the node blockchain...
        Mockito.verify(mockPeerClient, times(1)).getPeerBlockchain();
        Mockito.verify(mockBlockchain, times(1)).replaceChain(any());
        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Attempting to synchronize with the following peer: [{}]", logsList.get(2).getMessage());
        assertEquals("failure - incorrect logging message displayed","Could not synchronize with the following peer: [{}]. Reason: {}", logsList.get(3).getMessage());
    }

    @Test
    public void testRunPeerIsBehindNode_NegativeChainDiff() throws IOException, JSONException {
        //Given
        Mockito.when(mockPeer.getPeerStatus()).thenReturn(Status.UP);

        Mockito.when(mockPeerClient.getHealthySocketsList()).thenReturn("");
        Mockito.when(mockPeerClient.getPeerBlockchainSize()).thenReturn(-1); //will result in negative diff

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeerPollingService.run();

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        //Verify log message
        assertEquals("failure - incorrect logging message displayed","Peer [{}] is behind this node", logsList.get(2).getMessage());
    }
}
```

Listing I.11: PeerPollingServiceTest test class

```
class JsonBlockUtilTest {

    @Test
    void getBlockFromJsonObject() throws JSONException {
        //Given
        JSONObject jsonBlock = new JSONObject();

        jsonBlock.put("hash", "this");
        jsonBlock.put("previousHash", "is");
        jsonBlock.put("data", "a");
        jsonBlock.put("timeStamp", "test");
        jsonBlock.put("nonce", "json");
        jsonBlock.put("difficulty", "block");
        jsonBlock.put("proofOfWork", "=DDD");

        //When
        Block testBlock = JsonBlockUtil.getBlockFromJsonObject(jsonBlock);

        //Then
        assertEquals("this", testBlock.getHash());
        assertEquals("is", testBlock.getPreviousHash());
        assertEquals("a", testBlock.getData());
```

```
        assertEquals("test", testBlock.getTimeStamp());
        assertEquals("json", testBlock.getNonce());
        assertEquals("block", testBlock.getDifficulty());
        assertEquals("=DDD", testBlock.getProofOfWork());

        assertEquals("Block{hash='this', previousHash='is', data='a', timeStamp='test', nonce='
    json', difficulty='block', proofOfWork='=DDD'}", testBlock.toString());
    }

    @Test
    void getJsonObjectFromBlock() throws JSONException {
        //Given
        Block testBlock = new Block("this","is","a","test","block","lol","yeet");

        //When
        JSONObject jsonBlock = JsonBlockUtil.getJsonObjectFromBlock(testBlock);

        //Then
        assertEquals("this", jsonBlock.getString("hash"));
        assertEquals("is", jsonBlock.getString("previousHash"));
        assertEquals("a", jsonBlock.getString("data"));
        assertEquals("test", jsonBlock.getString("timeStamp"));
        assertEquals("block", jsonBlock.getString("nonce"));
        assertEquals("lol", jsonBlock.getString("difficulty"));
        assertEquals("yeet", jsonBlock.getString("proofOfWork"));

        assertEquals("{\"hash\":\"this\",\"previousHash\":\"is\",\"data\":\"a\",\"timeStamp\":\"
    test\",\"nonce\":\"block\",\"difficulty\":\"lol\",\"proofOfWork\":\"yeet\"}", jsonBlock.
    toString());
    }
}
```

Listing I.12: JsonBlockUtilTest test class

```
class PeerClientTest {
    PeerClient testPeerClient;

    @BeforeEach
    void setUp() {
        testPeerClient = new PeerClient("test:7090");
    }

    @Test
    void getPeerBlockchain() throws IOException, JSONException {
        //Given
        OkHttpClient mockOkHttpClient = mockHttpClient("[{\"hash\":\"89
    b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22\",\"previousHash\":\"dummyhash
    \",\"data\":\"dummydata\",\"timeStamp\":\"2020-11-07T19:40:57.585581100Z\",\"nonce\":\"
    dummydata\",\"difficulty\":\"3\",\"proofOfWork
    \":\"1101011101110100010010011001011010101001000010001011100011111011000110111000010010111111000100
    hash\":\"e1f6c84e1d746cbe7d20347ed8c233bb94d35f39728c1dc7109813a3184df3c2\",\"previousHash
    \":\"89b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22\",\"data\":\"poop\",\"
    timeStamp\":\"2021-01-11T17:20:45.196868Z\",\"nonce\":\"6\",\"difficulty\":\"2\",\"proofOfWork
    \":\"0011111101100101100001110100111111110010111111000000010010001100101001001001010000001110000001
    hash\":\"e8b4c47993c2b6aa71caee141f1dfb0f07c394f6cc963df53ee3ab13214d71e1\",\"previousHash
    \":\"e1f6c84e1d746cbe7d20347ed8c233bb94d35f39728c1dc7109813a3184df3c2\",\"data\":\"poodsadp
    \",\"timeStamp\":\"2021-01-11T17:20:45.587369900Z\",\"nonce\":\"23\",\"difficulty\":\"3\",\"
    proofOfWork
    \":\"0001000011110010101111111101110100100101010011111101011001111101000011101111001100001101111110
    hash\":\"596b27130c0af3539cc52dcb64561f29617a86e4db0890f020047eefa4917ed4\",\"previousHash
    \":\"e8b4c47993c2b6aa71caee141f1dfb0f07c394f6cc963df53ee3ab13214d71e1\",\"data\":\"pooasdasp
    \",\"timeStamp\":\"2021-01-11T17:20:45.665476200Z\",\"nonce\":\"3\",\"difficulty\":\"4\",\"
    proofOfWork
    \":\"0000110100110010001111111010011111111010110001100001100011001000110111110100010001101010101001
    ");
        ReflectionTestUtils.setField(testPeerClient, "client", mockOkHttpClient); //Create a mock
     client with a json blockchain response and inject into the test client

        //When
        ArrayList<Block> testResponse = testPeerClient.getPeerBlockchain();

        //Then
        assertTrue(new Blockchain(testResponse).isChainValid()); //Check the response is valid

        assertEquals(4, testResponse.size()); //Check its size

```

```
        assertEquals("Block{hash='89
    b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22', previousHash='dummyhash',
    data='dummydata', timeStamp='2020-11-07T19:40:57.585581100Z', nonce='dummydata', difficulty
    ='3', proofOfWork
    ='1101011101110100010010011001011010101001000010001011100011111011000110111000010010111111000100000
    ",
                testResponse.get(0).toString());
        assertEquals("Block{hash='
    e1f6c84e1d746cbe7d20347ed8c233bb94d35f39728c1dc7109813a3184df3c2', previousHash='89
    b76d274d54b62e56aea14299ea6feb282e5ba573cd378a42ecdfb00a772c22', data='poop', timeStamp
    ='2021-01-11T17:20:45.196868Z', nonce='6', difficulty='2', proofOfWork
    ='0011111101100101100001110100111111110010111111000000010010001100101001001001010000001110000010100
    ",
                testResponse.get(1).toString());
        assertEquals("Block{hash='
    e8b4c47993c2b6aa71caee141f1dfb0f07c394f6cc963df53ee3ab13214d71e1', previousHash='
    e1f6c84e1d746cbe7d20347ed8c233bb94d35f39728c1dc7109813a3184df3c2', data='poodsadp', timeStamp
    ='2021-01-11T17:20:45.587369900Z', nonce='23', difficulty='3', proofOfWork
    ='0001000011110010101111111101110100100101010011111101011001111101000011101111001100001101111110000
    ",
                testResponse.get(2).toString());
        assertEquals("Block{hash='596
    b27130c0af3539cc52dcb64561f29617a86e4db0890f020047eefa4917ed4', previousHash='
    e8b4c47993c2b6aa71caee141f1dfb0f07c394f6cc963df53ee3ab13214d71e1', data='pooasdasp', timeStamp
    ='2021-01-11T17:20:45.665476200Z', nonce='3', difficulty='4', proofOfWork
    ='0000110100110010001111110100111111101011000110000110001100100011011111010001000110101010100100111
    ",
                testResponse.get(3).toString());
    }

    @Test
    void getPeerBlockchainSize() throws IOException {
        //Given
        OkHttpClient mockOkHttpClient = mockHttpClient("69");
        ReflectionTestUtils.setField(testPeerClient, "client", mockOkHttpClient); //Create a mock
     client with a json blockchain size response and inject into the test client

        //When
        int testBlockchainSizeResponse = testPeerClient.getPeerBlockchainSize();

        //Then
        assertEquals(69, testBlockchainSizeResponse); //Check the correct value was returned
    }

    @Test
    void getPeerLastBlock() throws IOException, JSONException {
        //Given
        OkHttpClient mockOkHttpClient = mockHttpClient("{\"hash\":\"596
    b27130c0af3539cc52dcb64561f29617a86e4db0890f020047eefa4917ed4\",\"previousHash\":\"
    e8b4c47993c2b6aa71caee141f1dfb0f07c394f6cc963df53ee3ab13214d71e1\",\"data\":\"pooasdasp\",\"
    timeStamp\":\"2021-01-11T17:20:45.665476200Z\",\"nonce\":\"3\",\"difficulty\":\"4\",\"
    proofOfWork
    \":\"0000110100110010001111110100111111101011000110000110001100100011011111010001000110101010100100
    ");
        ReflectionTestUtils.setField(testPeerClient, "client", mockOkHttpClient); //Create a mock
     client with a json block response and inject into the test client

        //When
        Block testBlockchainLastBlockResponse = testPeerClient.getPeerLastBlock();

        //Then
        assertEquals("Block{hash='596
    b27130c0af3539cc52dcb64561f29617a86e4db0890f020047eefa4917ed4', previousHash='
    e8b4c47993c2b6aa71caee141f1dfb0f07c394f6cc963df53ee3ab13214d71e1', data='pooasdasp', timeStamp
    ='2021-01-11T17:20:45.665476200Z', nonce='3', difficulty='4', proofOfWork
    ='0000110100110010001111110100111111101011000110000110001100100011011111010001000110101010100100111
    ",
                testBlockchainLastBlockResponse.toString());
    }

    @Test
    void getPeerHealth() throws IOException, JSONException {
        //Given
        OkHttpClient mockOkHttpClient = mockHttpClient("{\"status\":\"UP\"}");
        ReflectionTestUtils.setField(testPeerClient, "client", mockOkHttpClient); //Create a mock
     client with a json health status and inject into the test client
```

```
        //When
        String testPeerStatusResponse = testPeerClient.getPeerHealth();

        //Then
        assertEquals("UP", testPeerStatusResponse);
    }

    @Test
    void addBlockToPeer() throws IOException, JSONException {
        //Given
        OkHttpClient mockOkHttpClient = mockHttpClient("Block added successfully!");
        ReflectionTestUtils.setField(testPeerClient, "client", mockOkHttpClient); //The response for this call will either be success, or invalid object exception message

        //When
        String testAddBlockResponse = testPeerClient.addBlockToPeer(new Block().genesis());

        //Then
        assertEquals("Block added successfully!", testAddBlockResponse);
    }

    @Test
    void getHealthySocketsList() throws IOException, JSONException {
        //Given
        OkHttpClient mockOkHttpClient = mockHttpClient("[{\"peerSocket\":\"54.90.44.155:8080\",\"peerStatus\":\"UNKNOWN\"},{\"peerSocket\":\"85.123.44.55:8080\",\"peerStatus\":\"UNKNOWN\"},{\"peerSocket\":\"33.44.55.66:6666\",\"peerStatus\":\"UP\"},{\"peerSocket\":\"123.24.53.42:8080\",\"peerStatus\":\"UP\"},{\"peerSocket\":\"69.42.0.88:3636\",\"peerStatus\":\"UNKNOWN\"}]");
        ReflectionTestUtils.setField(testPeerClient, "client", mockOkHttpClient); //Create a mock client with a json health status and inject into the test client

        //When
        String testHealthySocketsListReponse = testPeerClient.getHealthySocketsList();

        //Then
        assertEquals("33.44.55.66:6666,123.24.53.42:8080", testHealthySocketsListReponse);
    }

    private static OkHttpClient mockHttpClient(final String serializedBody) throws IOException {
        final OkHttpClient okHttpClient = Mockito.mock(OkHttpClient.class);

        final Call remoteCall = Mockito.mock(Call.class);

        final Response response = new Response.Builder()
                .request(new Request.Builder().url("http://url.com").build())
                .protocol(Protocol.HTTP_1_1)
                .code(200).message("").body(
                        ResponseBody.create(
                                MediaType.parse("application/json"),
                                serializedBody
                        ))
                .build();

        Mockito.when(remoteCall.execute()).thenReturn(response);
        Mockito.when(okHttpClient.newCall(any())).thenReturn(remoteCall);

        return okHttpClient;
    }
}
```

Listing I.13: PeerClientTest test class

```
class PeerTest {
    @Mock
    PeerPollingService mockPeerPollingService;

    @Mock
    PeerBroadcastingService mockPeerBroadcastingService;

    @BeforeEach
    void setUp() {
        //Given
        MockitoAnnotations.initMocks(this);
    }
```

```
13
14      @Test
15      void testPeerInitialisation(){
16          //Given
17          Peer testPeer;
18
19          //When
20          testPeer = new Peer("10.100.156.55:7090", new ScheduledThreadPoolExecutor(1));
21
22          //Then
23          assertEquals(Status.UNKNOWN, testPeer.getPeerStatus());
24          assertEquals("10.100.156.55:7090", testPeer.getPeerSocket());
25      }
26
27      @Test
28      void testPeerPolling(){
29          //Given
30          Peer testPeer = new Peer("10.100.156.55:7090", new ScheduledThreadPoolExecutor(1));
31          ReflectionTestUtils.setField(testPeer, "peerPoller", mockPeerPollingService);
32
33          //When
34          testPeer.startPolling();
35
36          //Then
37          Mockito.verify(mockPeerPollingService, times(1)).start();
38      }
39
40      @Test
41      void testPeerBroadcasting(){
42          //Given
43          Peer testPeer = new Peer("10.100.156.55:7090", new ScheduledThreadPoolExecutor(1));
44          ReflectionTestUtils.setField(testPeer, "peerBroadcaster", mockPeerBroadcastingService);
45
46          Block testBlock = new Block("this","is","a","test","block","lol","yeet");;
47
48          //When
49          testPeer.broadcastBlock(testBlock);
50
51          //Then
52          Mockito.verify(mockPeerBroadcastingService, times(1)).broadcastBlock(testBlock);
53      }
54
55      @Test
56      void testPeerStatusGetterAndSetter(){
57          //Given
58          Peer testPeer = new Peer("10.100.156.55:7090", new ScheduledThreadPoolExecutor(1));
59          ReflectionTestUtils.setField(testPeer, "peerBroadcaster", mockPeerBroadcastingService);
60
61          //When
62          testPeer.setPeerStatus(Status.DOWN);
63
64          //Then
65          assertEquals(Status.DOWN, testPeer.getPeerStatus());
66      }
67  }
```

Listing I.14: PeerTest test class

```
1   class PeersTest {
2       Logger logger = (Logger) LoggerFactory.getLogger(Peers.class);
3       ListAppender<ILoggingEvent> listAppender = new ListAppender<>();
4
5       @Mock
6       Peer mockPeer;
7
8       @BeforeEach
9       void setUp() {
10          //Given
11          MockitoAnnotations.initMocks(this);
12      }
13
14      @Test
15      void testPeerInitialisationAndGetPeerList() {
16          //Given
17          Peers testPeers;
18
```

```
        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeers = new Peers("10.10.10.10:5000", 1000, "20.20.20.20:3000,30.30.30.30:8000");

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals(2, testPeers.getPeerList().size());
        assertEquals("20.20.20.20:3000", testPeers.getPeerList().get(0).getPeerSocket());
        assertEquals("30.30.30.30:8000", testPeers.getPeerList().get(1).getPeerSocket());

        assertEquals("10.10.10.10:5000", ReflectionTestUtils.getField(testPeers, "NODE_SOCKET"));
        assertEquals(1000, ReflectionTestUtils.getField(testPeers, "MAX_PEERS"));

        //NOTE: the below assertion could fail if the system running this test has more than 1000 logical cores, highly unlikely...
        assertEquals("failure - incorrect logging message displayed","It is recommended to set the max peers to less than {} for your system, performance may be impacted...", logsList.get(0).getMessage());
    }

    @Test
    void testAddSocketMaxPeerExceeded(){
        //Given
        Peers testPeers = new Peers("10.10.10.10:5000", 2, "20.20.20.20:3000");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeers.addSocketsList("30.30.30.30:8000,58.23.15.55:9898");

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - incorrect logging message displayed","Unable to add new peer [{}] as max peer size of {} has been reached", logsList.get(2).getMessage());
    }

    @Test
    void testAddSocketSameNodeSocket(){
        //Given
        Peers testPeers = new Peers("10.10.10.10:5000", 2, "20.20.20.20:3000");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeers.addSocketsList("10.10.10.10:5000");

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - incorrect logging message displayed","Unable to add new peer [{}] as its socket refers to this node!", logsList.get(1).getMessage());
    }

    @Test
    void testAddSocketInvalidSocket(){
        //Given
        Peers testPeers = new Peers("10.10.10.10:5000", 2, "20.20.20.20:3000");

        //When
        listAppender.start();
        logger.addAppender(listAppender); //start log capture...

        testPeers.addSocketsList("yeet:5000");

        //Then
        List<ILoggingEvent> logsList = listAppender.list; //...store captured logs

        assertEquals("failure - incorrect logging message displayed","Unable to add new peer [{}] as its socket is of invalid format", logsList.get(1).getMessage());
    }
```

```
89
90      @Test
91      void testAddSocketKnownSocket(){
92          //Given
93          Peers testPeers = new Peers("10.10.10.10:5000", 2, "20.20.20.20:3000");
94
95          //When
96          listAppender.start();
97          logger.addAppender(listAppender); //start log capture...
98
99          testPeers.addSocketsList("20.20.20.20:3000");
100
101         //Then
102         List<ILoggingEvent> logsList = listAppender.list; //...store captured logs
103
104         assertEquals("failure - incorrect logging message displayed","Unable to add new peer [{}]
        as its already known", logsList.get(1).getMessage());
105     }
106
107     @Test
108     void testBroadcastBlockToPeers() {
109         //Given
110         Peers testPeers = new Peers("", 2, "");
111
112         ArrayList<Peer> testPeerList = new ArrayList<>();
113         testPeerList.add(mockPeer); //Create a dummy peerList with the mock peer...
114         ReflectionTestUtils.setField(testPeers, "peerList", testPeerList); //...And inject it
        into testPeers
115
116         Block testBlock = new Block("this","is","a","test","block","lol","yeet");;
117
118         //When
119         testPeers.broadcastBlockToPeers(testBlock);
120
121         //Then
122         Mockito.verify(mockPeer, times(1)).broadcastBlock(testBlock);
123     }
124 }
```

Listing I.15: PeersTest test class

```
1  class JrcServerConfigTest {
2
3      Logger logger = (Logger) LoggerFactory.getLogger(Blockchain.class);
4      ListAppender<ILoggingEvent> listAppender = new ListAppender<>();
5
6      @Test
7      public void testServerConfigInitialisesBlockchain() throws InvalidObjectException {
8          //Given
9          JrcServerConfig testConfig = new JrcServerConfig();
10         Block genesisBlock = new Block().genesis();
11
12         //When
13         listAppender.start();
14         logger.addAppender(listAppender); //start log capture...
15
16         Blockchain testChain = testConfig.blockchain();
17         Boolean isChainValid = testChain.isChainValid();
18
19         //Then
20         List<ILoggingEvent> logsList = listAppender.list; //...store captured logs
21
22         assertEquals("failure - hash of the first block in the blockchain does not equal the
       genesis hash", genesisBlock.getHash(), testChain.getChain().get(0).getHash());
23         assertEquals("failure - data of the first block in the blockchain does not equal the
       genesis data", genesisBlock.getData(), testChain.getChain().get(0).getData());
24         assertEquals("failure - previous hash of the first block in the blockchain does not equal
        the genesis previous hash", genesisBlock.getPreviousHash(), testChain.getChain().get(0).
       getPreviousHash());
25         assertEquals("failure - timestamp of the first block in the blockchain does not equal the
        genesis timestamp", genesisBlock.getTimeStamp(), testChain.getChain().get(0).getTimeStamp());
26
27         assertTrue("failure - valid chain incorrectly flagged as invalid", isChainValid);
28
29         assertEquals("failure - Original blockchains chain is incorrect length", 1, testChain.
       getChain().size());
```

```
30
31          assertEquals("failure - incorrect logging message displayed","Attempting to initialise a
    blockchain with the following chain array: {}...", logsList.get(0).getMessage());
32          assertEquals("failure - incorrect logging message displayed","A Fresh blockchain has been
     initialised with genesis block...", logsList.get(1).getMessage());
33      }
34
35      @Test
36      public void testServerConfigInitialisesBlockchainApiDelegateImpl(){
37          //Given
38          JrcServerConfig testConfig = new JrcServerConfig();
39
40          //When
41          BlockchainApiDelegateImpl testBlockchainApiDelegateImpl = testConfig.
    blockchainApiDelegateImpl();
42
43          //Then
44          assertNotNull(testBlockchainApiDelegateImpl);
45      }
46
47      @Test
48      public void testServerConfigInitialisesPeers(){
49          //Given
50          JrcServerConfig testConfig = new JrcServerConfig();
51          ReflectionTestUtils.setField(testConfig, "NODE_SOCKET", "127.0.0.1:2323");
52          ReflectionTestUtils.setField(testConfig, "PEERS_MAX", 200);
53          ReflectionTestUtils.setField(testConfig, "PEERS_SOCKETS", "
    54.90.44.155:8080,23.444.23.145:2020");
54
55          //When
56          Peers testPeers = testConfig.peers();
57
58          //Then
59          assertEquals("failure - Node Socket value incorrect", "127.0.0.1:2323",
    ReflectionTestUtils.getField(testPeers, "NODE_SOCKET"));
60          assertEquals("failure - Max peers value incorrect", 200, ReflectionTestUtils.getField(
    testPeers, "MAX_PEERS"));
61          assertEquals("failure - Max peers value incorrect", "54.90.44.155:8080", testPeers.
    getPeerList().get(0).getPeerSocket());
62          assertEquals("failure - Max peers value incorrect", "23.444.23.145:2020", testPeers.
    getPeerList().get(1).getPeerSocket());
63      }
64
65      @Test
66      public void testServerConfigInitialisesPeersApiDelegateImpl(){
67          //Given
68          JrcServerConfig testConfig = new JrcServerConfig();
69
70          //When
71          PeersApiDelegateImpl testPeersApiDelegateImpl = testConfig.peersApiDelegateImpl();
72
73          //Then
74          assertNotNull(testPeersApiDelegateImpl);
75      }
76  }
```

Listing I.16: JrcServerConfigTest test class

Appendix J

Minerate Integration Test

```java
class MinerateTest {
    private final Thread thisThread = Thread.currentThread();
    private final int timeToRun = 1800000; // 30 minutes;

    //This test will add blocks to the chain for 30 mins inorder to test difficulty is adjusted correctly around the minerate, turn logging level to info in logback.xml so console isn't flooded
    @Test
    public void testMinerate() throws InvalidObjectException{
        Blockchain testChain = new Blockchain(new ArrayList<>()); //Initialise new blockchain

        new Thread(new Runnable() {
            @SneakyThrows
            public void run() {
                sleep(timeToRun);
                thisThread.interrupt();
            }
        }).start(); //Sleep current thread for timeToRun ms

        while (!Thread.interrupted()) { //Loop whilst thread is interrupted
            Block nextBlock = new Block().mineBlock(testChain.getLastBlock(), getRandomData());

            try {
                testChain.addBlock(nextBlock); //Add a block with random data
            } catch (NullPointerException e) {
                //Since there is no peers bean, the peers.broadcastBlockToPeers() method call will fail, catch this and fail silently as its not relevant to this test
            }

            long diffSeconds = BlockUtil.calcBlockTimeDiff(testChain.getLastBlock().getTimeStamp(),testChain.getChain().get(testChain.getChain().size() - 2).getTimeStamp()); //Difference in seconds between the block currently being mined, and the previously mined block

            System.out.println("Block took: " + diffSeconds + " seconds to mine...");
        }

        System.out.println("Blocks should take on average 120 seconds to mine as determined by the minerate, check the above console logs!");
    }

    private String getRandomData(){
        byte[] array = new byte[7];
        new Random().nextBytes(array);

        return new String(array, StandardCharsets.UTF_8);
    }
}
```