Kan, Yin Yee
University College London

Under the supervision of
Sarah Meiklejohn

September 12, 2016

CRYPTOMAGIC
Creating money with cryptography

This report is submitted as part requirement for the MSc in computer science at the University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

The first half of the paper aims to provide an introduction to the technical underpinnings of cryptocurrencies that is accessible to readers who may not have a background in computer science. Section 1 briefly explains how cryptocurrencies remove the need for central authorities and intermediaries in the financial system. Section 2 is a primer on the two cryptographic primitives that form the backbone of a cryptocurrency: cryptographic hash functions and digital signatures. Section 3 provides an illustration of data structures that are built using those cryptographic primitives. Section 4 presents both the client-server and peer-to-peer models of networking for the sake of contrast, since the client-server model is at odds with the goal of decentralisation. The accepted usage of the term "consensus" encompasses both the double-spend prevention problem as well as the classical problem of coordinating agreement in distributed systems; however, this paper finds it helpful to distinguish between the problems. Section 5 explains how a lottery provides security against double-spending in a cryptocurrency system and how consensus is achieved by using a property of the blockchain as an oracle.

The second half of the paper presents the implementation of discōCoin: a proof-of-work cryptocurrency built from scratch using only Java standard libraries. The design of discōCoin is based in large part on Bitcoin. Section 6 explains the types of nodes that can connect to the discōCoin network and the messaging protocol used for communication between the nodes. It concludes with a description of the features that are planned for the next development phase of discōCoin.

Contents

List of Figures

1 Introduction

Intermediaries and central authorities perform many critical roles within the financial system. Central banks manage the money supply and prevent counterfeiting. Traditional banks take on the risk of transforming liquid short-term deposits into useful long-term capital. Depositories maintain records of transactions and balances in order to prevent double-spending. Clearing banks enable payments by linking networks of accounts held at different banks.

Replacing intermediaries with cryptography

Can a financial system function without intermediaries or central authorities? Cryptocurrencies are an ongoing experiment in decentralised financial systems. In proof-of-work systems, new currency is minted and distributed through a process known as mining. Counterfeiting is impossible as cryptocurrencies are built using unforgeable cryptographic primitives. Any interested party can become a participant of equal standing in a crypto-financial system. There are no intermediaries or authorities with whom accounts are maintained and identities established. Double-spending is prevented because every participant has a copy of all transactions ever made, in chronological order, from inception to the present time. Payments denominated in the same cryptocurrency are made directly from sender to recipient since the network is not divided into sub-networks depending on which intermediary the participants have an account with [1].

2 Cryptographic primitives

The primitives used to construct cryptocurrencies are cryptographic hash functions and digital signatures. Both these primitives in turn rely on one-way functions. One-way functions are efficiently computable in one direction but computationally infeasible to invert [2]. This results in a property known as hiding.

Hiding Knowledge of $f(x)$ does not reveal any information about the value of x.

2.1 Cryptographic hash functions

An important property of cryptographic hash functions is collision-resistance.

Collision-resistance For a given cryptographic hash function h, it is computationally infeasible to find a pair of inputs (x, y) such that $h(x) = h(y)$ for $x \neq y$.

Collision-resistance is useful for making and enforcing commitments [3].

Non-repudiability If Alice makes a commitment to x to Bob and makes $h(x)$ public knowledge, she cannot later claim to have instead made a commitment to y since $h(y) \neq h(x)$.

Unforgeability Similarly, Bob cannot pretend that Alice commited to y instead by forging $h(y) = h(x)$.

Collision-resistance also means that $h(x)$ can be used as a unique identifier of x; in other words, a value can alternatively be represented by its hash [2].

Another useful design feature of hash functions in general is the ability to take a variable-sized input and produce a fixed-size output [3].

2.2 Digital signatures

Digital signatures are built using public key cryptography, which is also known as asymmetric cryptography.

Public key cryptography

Despite its name, public key cryptography in fact uses a *pair* of public and private keys. The pair of public and private keys are unique to each other: there is exactly one private key for each public key, and vice versa. As the names suggest, the public key is made widely known, while the private key is kept secret.

In order for the system to be secure, knowledge of the public key must not leak any information about the private key (*see* **hiding**). Public key cryptography makes use of a special category of one-way functions called trapdoor functions. Trapdoor functions can be efficiently inverted provided one has knowledge of the secret trapdoor [2] [3].

If Alice wants to be certain that her messages to Bob can only be read by him, she can encrypt her messages using Bob's public key. Bob then uses his private key to decrypt the message sent by Alice before reading it. Bob can efficiently reverse Alice's encryption because his private key is a trapdoor [2].

Encryption $$\text{ciphertext} = E_{public}(\text{plaintext}) \tag{1}$$

Decryption $$\text{plaintext} = D_{private}(\text{ciphertext}) \tag{2}$$

An additional requirement for digital signatures is that the encryption and decryption processes are one-way trapdoor permutations [4], where

Permutation $$D_{private}(E_{public}(\text{plaintext})) = E_{public}(D_{private}(\text{plaintext})) \tag{3}$$

Then, if Bob wishes to prove that he authored a message, he can publish a signature together with the message [4].

$$\textbf{Signing a message} \qquad \text{signature} = S_{private}(\text{message}) \tag{4}$$

Anyone can then verify that the message is authentic by using Bob's public key to check if the message can be recovered [4].

$$\textbf{Verifying a signature} \qquad \text{message} = V_{public}(\text{signature}) \tag{5}$$

3 Data structures

3.1 Transactions

As with regular currencies, transactions transfer ownership of cryptocurrency from one account to another, where an account is a public key.

Non-mint transactions can have any number of inputs and outputs. An output is associated with the public key of its owner while an input is simply a pointer to an unspent output from a previous transaction. An input must be accompanied by a signature. A valid signature proves that the person attempting to spend the input knows the private key that corresponds to the public key of the output that the input references. Therefore transactions are simply a series of **digital signatures** [1].

Mint transactions
Mint transactions are special transactions which create new units of cryptocurrency; they do not have any inputs. The participant holding a right to mint also has the right to elect the recipient of the newly-minted currency. The minting protocol functions as the monetary policy of a cryptocurrency [1].

Provenance Starting from any input (or, equivalently, output), a chain of transactions can be traced back to its minting.

Since all transactions are public and completely traceable, the basic privacy afforded to participants in a cryptocurrency system arises from the separation of accounts from identities. A participant can have an unlimited number of accounts; each account is a pseudonym for the account holder.

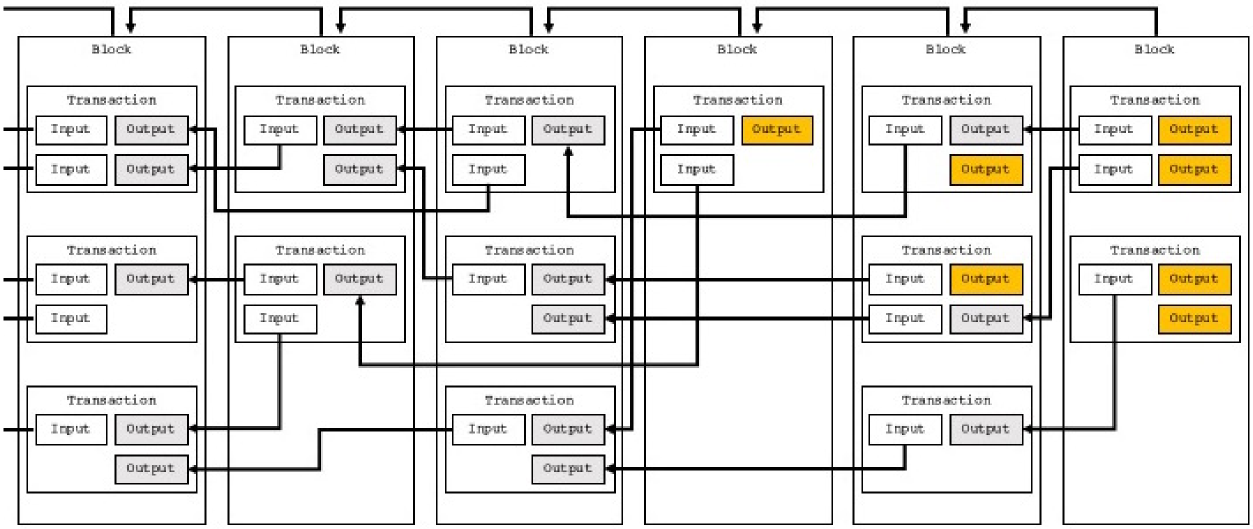

Figure 1: Every `Input` references an unspent `Output`. Spent `Outputs` are in grey; unspent in yellow.

Pseudonymity Although accounts and transactions are public knowledge, the identities of the account holders are known only to the account holders themselves[1].

In practice, the veil of pseudonymity can be pierced using data analysis [5]. It is considered good practice use a new public key for each transaction. Additional measures can be taken to bolster privacy such as pooling transactions through a mixing service so that unrelated transaction inputs and outputs are scrambled together [6].

3.2 Merkle trees

It is possible for each block to contain only one transaction. However, this leads to an impractically low transaction throughput hence transactions are gathered into blocks. By design, the average throughput of blocks is fixed while the number of transactions per block can vary.

$$\text{transaction throughput} = \underset{\text{(variable)}}{\text{number of transactions per block}} \times \underset{\text{(fixed)}}{\text{block throughput}}$$

Merkle trees are used to obtain a compact fixed-length commitment to the transactions which are included in a block. Transactions are arranged at the leaves of a binary tree. The first

[1]There is nothing to prevent account holders making their identities known if they wish to do so. For instance, a seller would have to provide a recipient account to their customers in order to receive payment.

level of nodes contains the hashes of the individual leaves. At each stage of compression, the right and left child nodes are concatenated and the hash of the concatenation is designated as the parent node. Compression is recursively performed until there is a single parent node remaining – this node is known as the root of the Merkle tree. The root functions as a commitment to the entire collection of transactions and is included in the pre-hash value of the block [7]. Since the root of the Merkle tree is obtained through repeated hashing, the transactions cannot be rebuilt from the final root hence they must be included in the block.

3.3 Blockchain

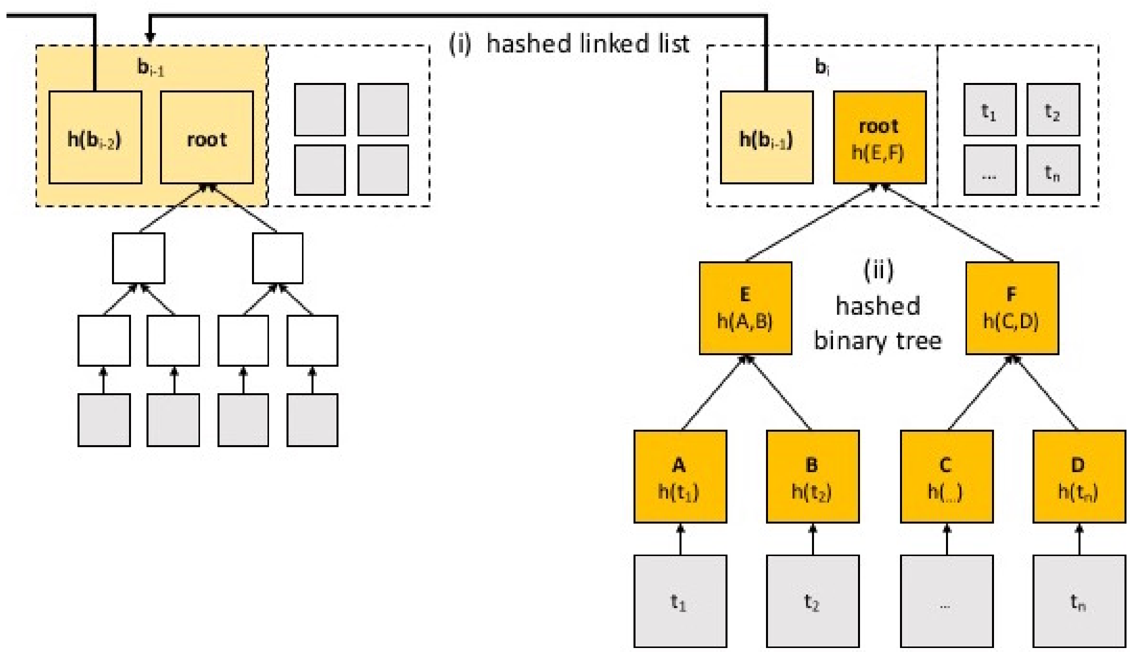

Figure 2: Cryptocurrency data structures (i) hashed linked list (*aka* blockchain) (ii) hashed binary tree (*aka* Merkle tree) [3]

A blockchain is a hashed linked list of blocks. Each block is uniquely identified by its hash and has a pointer to the hash of the previous block. However, the blockchain data structure employs a much stronger method of ensuring a strict ordering of information: the hash of the previous block is included in the pre-hash value of the current block. Any attempt to re-order blocks will change the hashes and therefore be detected [3].

Hashing a block $$\text{block hash} = h(\text{previous block's hash} \mid \text{root of Merkle tree}) \quad (6)$$

where | denotes concatenation.

Genesis block

The genesis block is the first block in the blockchain. Since it has no predecessor, it is exempted from the requirement to include the previous block's hash[2].

Blockchain as a state machine

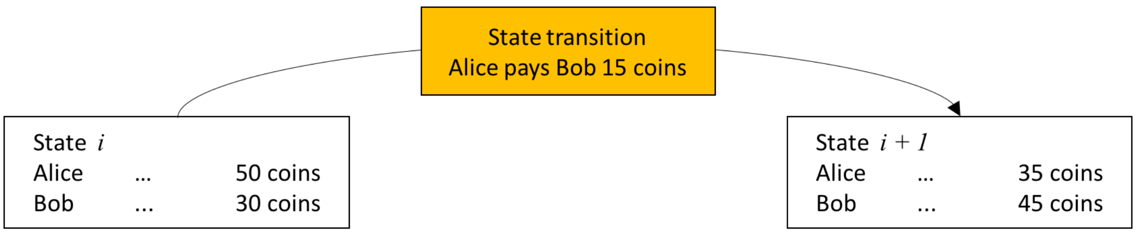

Figure 3: A valid state transition causes the blockchain to move from state i to $i + 1$ [9].

The blockchain is a history-preserving state machine where transactions are the state transitions [3]. The current state is a record of the addresses that own the current unspent outputs. The total value of unspent outputs is always equal to the current total value of the monetary supply.

[2]Bitcoin's genesis block hash includes the following text: "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks" [8]. In keeping with tradition, discōCoin's genesis block replaces the hash of the previous block with the hash of a secret string.

4 Networking

In order to be able to communicate with other computers, a computer must be connected to those computers. A network is a group of computers which are connected to each other; the Internet is a network of networks.

4.1 Client-server model

Most of the communication that takes place over the Internet is based on the client-server model, where the client sends a request to the server for information which is stored on the server. In this model, the information is only available when the server is powered on and connected to the network. The model is vulnerable to denial-of-service attacks that flood the server with requests so that legitimate clients are unable to connect to the server[3].

The client-server model has a high degree of centralisation which renders it unsuitable for cryptocurrency systems.

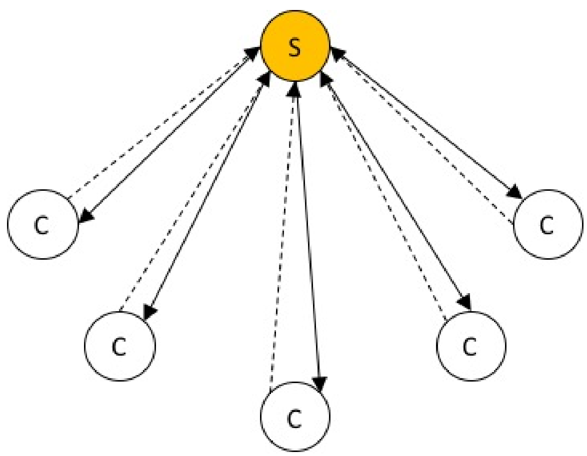

Figure 4: Illustration of the client-server model. The dotted arrows represent requests from clients to the server; solid arrows represent information from server to client.

4.2 Peer-to-peer model

The peer-to-peer model can be thought of as a leaderless organisation. A peer can be simultaneously a client and a server. Autonomy is the defining characteristic of peer-to-peer networks. Peers are free to join and leave the network as they wish. Each peer decides for itself if it will act as a server and the information that it is willing to serve.

[3]Peer-to-peer networks are not immune to denial-of-service attacks [10]

Multiple peers can decide to act as servers for the same piece of information, resulting in that piece of information being replicated and stored in different parts of the network [11]. The resilience of a particular piece of information to denial-of-service attacks increases as the number of peers who choose to act as servers for that piece of information grows [12]. This works well for information which remains static, such as songs or movies, but presents a challenge for dynamic information, such as bank account balances.

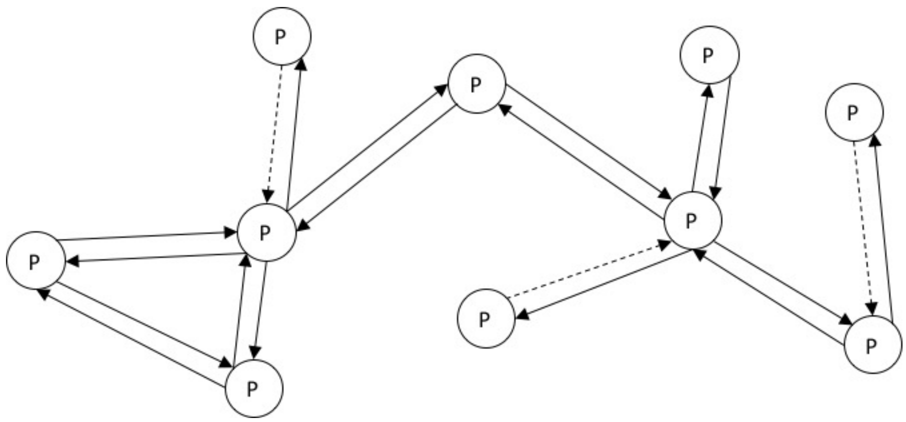

Figure 5: Illustration of the peer-to-peer model. A dotted-solid pair of arrows represents a one-sided client-server relationship; a pair of solid arrows represents a symbiotic relationship.

5 Security and consensus

The distributed nature of peer-to-peer networks presents two interesting problems for cryptocurrency systems: (1) preventing dishonest peers from double-spending and (2) agreeing on a single version of the truth.

A note on the definition of "consensus" in this paper. The accepted usage of the term encapsulates both problems, however this paper finds it helpful to make a distinction and will use "security" to refer to the double-spend prevention problem and "consensus" to refer to the problem of selecting a canonical chain.

5.1 Security

Transactions can originate from any node in the network and are relayed by broadcast to as many nodes as possible. A transaction is sucessful if it is included in a block on the canonical chain. However, there is no central authority that is responsible for ensuring that no dishonest transactions are included in a block. The solution is to hold a lottery for the creation of each new block [13].

The lottery confers two related privileges to the winner. The first is the power to determine which transactions are included in the block. We will asume that, in addition to not attempting to transact dishonestly, honest nodes will not include dishonest transactions in the blocks that they create. Suppose that Alice wins the lottery and includes a transaction in which she spends an output that has already been spent. To prevent Alice from getting away with double-spending, the block that she creates (including the transactions in the block) is checked by the other nodes in the network.

In order to create an incentive for other nodes to check Alice's block, the lottery winner also wins the right to create a **mint transaction** [14], [15]. If Alice's deception is discovered, her block is invalidated and another node that is able to present a valid rival block becomes the new lottery winner. The more participants in the lottery, the less likely it is for any one participant to win, which increases the security of the cryptocurrency system.

Mining Participation in the block lottery.

5.1.1 Proof-of-work

Proof-of-work is the most common implementation of the block lottery. It involves finding a **hash** for the block that meets certain requirements. Most commonly, the requirement is that the hash must be smaller in value than a target threshold[4].

The output of a hash function is random: hash values are distributed uniformly over the entire output space. Placing restrictions on the definition of a valid hash therefore results in a reduction in the probability space. The smaller the probability space, the more "difficult" it is to find a valid hash. The random nature of the output values means that there is no shortcut to finding a hash that meets the requirements, hence each attempt to find a valid hash is akin to purchasing a lottery ticket. As with all lotteries, although there is the element of luck, the chances of winning increase with the number of lottery tickets purchased [3].

The rate at which a node computes the hash function is called the hash rate. As nodes join or leave the network, the cumulative hash rate of the system increases or decreases. The target threshold can be periodically adjusted so that the average block throughput rate remains roughly constant.

[4] Equivalently, the requirement could be for the hash to be *larger* than the target threshold.

5.1.2 Proof-of-stake

Proof-of-stake is an alternative lottery design in which lottery tickets are in the form of ownership of the cryptocurrency. Proof-of-stake is based on the argument that the larger the stakeholder, the more incentive it naturally has to protect the chain from attacks which could result in the loss of trust in the system [16], [17]. Besides rewarding the rich, proof-of-stake also introduces the risk of censorship where large stakeholders could prevent transactions from being included in blocks.

On the other hand, proof-of-stake could weaken the link between the mint reward and the incentive to secure the cryptocurrency from attacks. In theory, this could mean much smaller mint rewards or even a complete break between the distribution of newly-minted currency and the process of securing the blockchain [18].

5.1.3 Other lottery designs

The lottery design can be broken down into two parts: defining a valid lottery ticket and implementing an algorithm to select exactly one winning ticket for each round using an algorithm which is verifiably random [3]. A naive design would be to define as a valid lottery ticket to be an IP address. This design can easily be compromised through Sybil attacks[5] [1]. Some other designs which have been proposed are proof-of-activity [17] and proof-of-space [19].

5.2 Consensus

For the blockchain to function as a deterministic state machine, there can only be one canonical ordering of blocks. However, each node in a peer-to-peer network has to determine which sequence of state transitions it accepts as the canonical chain. Thus, achieving consensus is fundamentally a problem of coordination.

5.2.1 Paxos

Paxos is a classic consensus protocol for distributed systems. The protocol seeks to ensure that the latest state transitions proposals are propagated throughout the network. State transition proposals are time-stamped and older proposals are dropped.

[5] Multiple accounts that are in fact controlled by the same party; in other words, one person masquarading as a crowd.

Any node can propose a state transition. For ease of explanation, we assume that state transitions are numbered and that lower numbered state transitions are older. If Alice wants to propose a state transition, she first sends the number of the state transition that she wishes to propose [20].

Scenario 1
Upon receiving Alice's proposal, if the number of the state transition that Alice has proposed is the highest one that he has received, Bob makes a record of Alice's proposed state transition number and gives Alice the go-ahead. Alice then sends Bob the value of her proposed state transition.

Scenario 2
However, if Bob has already received a higher state transition number proposal from Carol, Bob informs Alice of the value of the state transition proposed by Carol. Alice then echoes the value proposed by Carol. Nodes accept the highest-numbered proposal that it receives; a proposal that is accepted by the majority of nodes acquires the property of being chosen.

To ensure that there is progress made on accepting proposals, the protocol introduces the role of a coordinator. Nodes make their proposals to the coordinator who decides on the order of the state transitions and informs the nodes accordingly [20].

5.2.2 Two-phase commit

The goal of the two-phase commit protocol is ensure atomic commitment.

Atomic commitment *Either* all nodes must together update their state *or* no updates are made by any node.

If Alice, who has an account with Bank A, wants to pay Bob, whose account is with Bank B, it is clear that either Alice's account balance is reduced by the amount of the payment and Bob's account is increased by the same amount, or there should not be any change to either Alice's or Bob's accounts.

Similar to Paxos, a state transition is proposed to a coordinator. The coordinator sends the proposed state transition to all nodes that are involved in the transaction. Each node responds with a yes or no vote on the proposed state transition. If the coordinator receives a unanimous yes vote, it sends a confirmation to commit to all nodes; otherwise it sends an instruction to abort [21].

5.2.3 Blockchain consensus

The main disadvantage of the Paxos and two-phase commit protocols is centralisation in the form of a coordinator role. For large-scale networks, it is unfeasible for the coordinator to maintain a connection with every single node [11].

Consensus can be achieved in blockchain-based networks by using some property of contending chains as an oracle. Every node queries its peers for the selected property of their respective chains and selects for themselves the chain with the highest value as the canonical chain. One such property is chain height i.e. the number of blocks in a chain, hence the canonical chain is the longest chain.

Another property is chain weight. Recall that in a proof-of-work system, there is a difficulty measure associated with the target threshold for a valid hash which can change as the network grows or shrinks. Chain weight is the sum of the difficulty of each block in the chain; accordingly the heaviest chain is canonical.

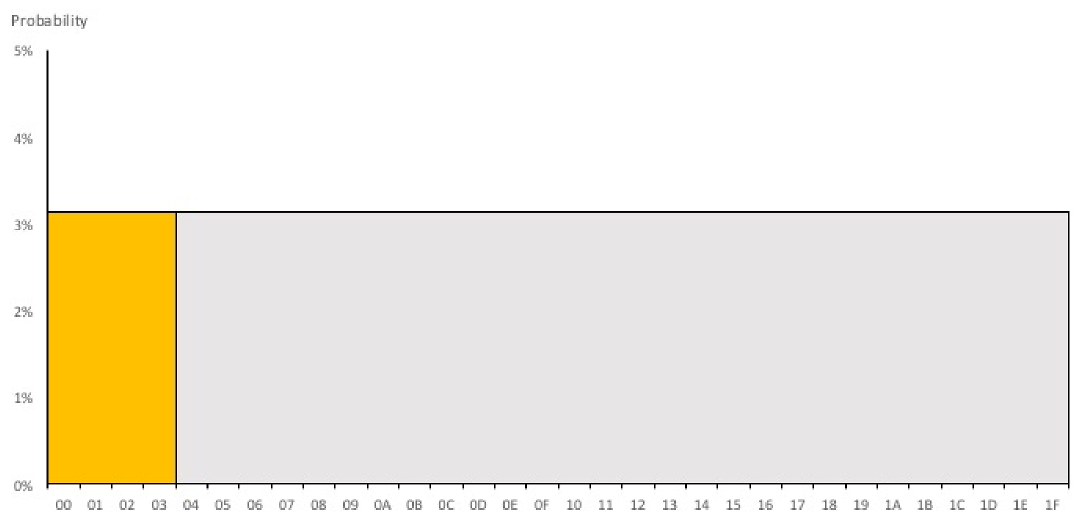

Figure 6: Uniform probability distribution over all possible hash values for a 32-bit output space. Restricting valid hash values to be smaller than 04 results in an eight-fold increase in difficulty. Although the hash function produces output over the entire space, only hash values found in the yellow area are considered valid.

5.3 Relationship between security and consensus

5.3.1 Byzantine agreement

The problem faced by the fictional Byzantine Generals is well-known in distributed computing. A group of generals intends to execute a coordinated attack on an enemy. The generals and their respective armies are encamped at a distance from each other and can only communicate by sending messages back and forth. However, some of the generals may be traitors who attempt to spread misinformation in the hopes that an uncoordinated attack will result in the loyal armies getting killed instead [22].

It is easy to see that a dishonest coordinator in the case of Paxos or two-phase commit could cause invalid state transitions to be included or non-atomic commitments to occur. To ensure that each node is able to discern the correct message, there must be $3f + 1$ coordinators in order for the system to be able to tolerate f faulty coordinators. Consider the case where $f = 1$: in a system with one faulty coordinator, an honest coordinator is only able to establish the truth if there are at least two other honest coordinators who can corroborate each other's message [22].

The protocols can then be modified such that instead of interacting with a single coordinator, nodes will interact (e.g. propose, vote) with as many coordinators as possible. The coordinators then engage in their own round of consensus: a coordinator must receive messages from $2f$ other coordinator nodes that are identical to its own in order to be certain that the message they have is correct; otherwise the message is discarded. Finally, each node will act on an instruction only if it has received the same instruction from $f + 1$ different coordinators [21].

5.3.2 Honest majority

Similarly, it is possible for the canonical chain to contain dishonest transactions if the block lottery is won sucessively by dishonest nodes. Given that the probability of finding a valid hash is proportional to the node's hash rate, in simplistic terms, a cryptocurrency that uses proof-of-work is secure if honest nodes control the majority of the hash power. Cryptocurrencies based on proof-of-stake are secure if the majority of the money supply is held by honest participants[6] [1].

[6] In fact, cryptocurrencies secured by proof-of-stake suffer from the "nothing-at-stake" problem where rational participants would be best served by helping to extend all competing chains [18].

6 discōCoin

discōCoin is a cryptocurrency written using Java standard libraries. Its design is based largely on Bitcoin. Transactions are signed using the RSA-512 digital signature scheme and the blockchain is secured by proof-of-work using the SHA-256 cryptographic hash function with chain height as the consensus oracle.

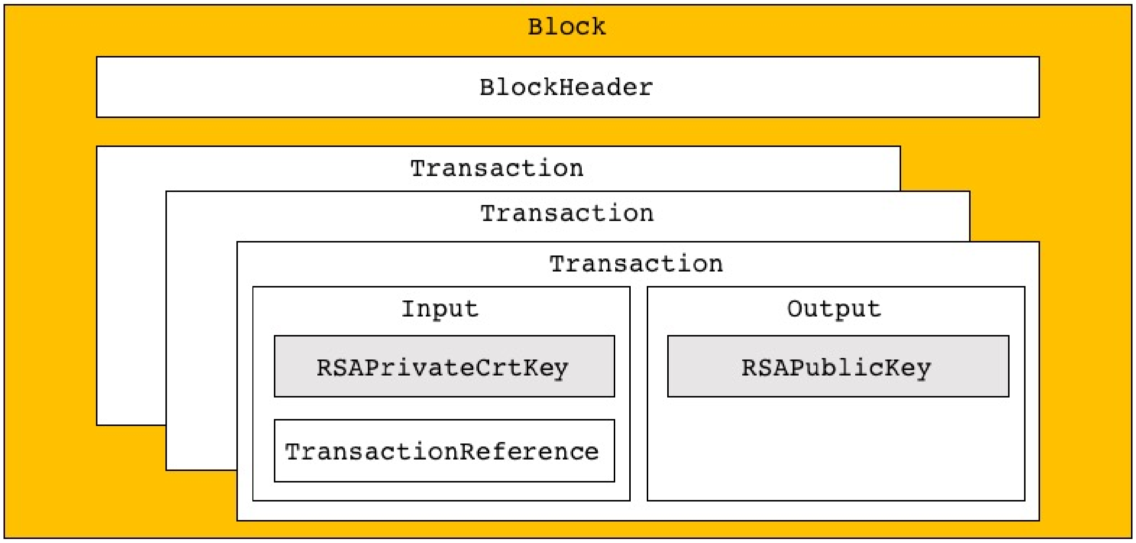

Figure 7: A discōCoin block and its components. Each box represents a Java class; grey boxes are from the Java standard library.

There are two types of nodes that can connect to the discōCoin network.

6.1 TransactionalNode

The `TransactionalNode` has a graphical user interface to facilitate creating and sending `Transactions`. A `TransactionalNode` can be launched on its own or together with an instance of the `NetworkNode` (*see* **user manual**). However, it must be connected to an active `NetworkNode` in order to send `Transactions` to the network since it does not itself mine `Blocks`.

Figure 8: Graphical user interface for the `TransactionalNode`. Acknowledgements are due to `http://www.freepik.com/`, `http://www.flaticon.com/` and `http://logomakr.com` for the discōCoin logo.

6.1.1 Creating a transaction

An empty `Transaction` is first created; `Inputs` and `Outputs` can subsequently be added or removed until the `Transaction` is finalised.

An `Input` consists of a `TransactionReference` and an `RSAPrivateKey`. A `TransactionReference` is a pointer to a previous `Output` on the blockchain.

Finalising a transaction

The process of finalising a `Transaction` begins by computing the transaction fee, which is the amount by which the sum of `Inputs` exceed the sum of `Outputs`. Transaction fees cannot be negative, for obvious reasons, although it can be zero – fees are gratuities akin to tipping a waiter. Next, each `Input` is checked against the latest list of unspent `Outputs`. If the `Input` is a valid unspent `Output`, its `RSAPrivateKey` is used to sign *all* `Outputs` to the `Transaction` (*see* **signing a message**).

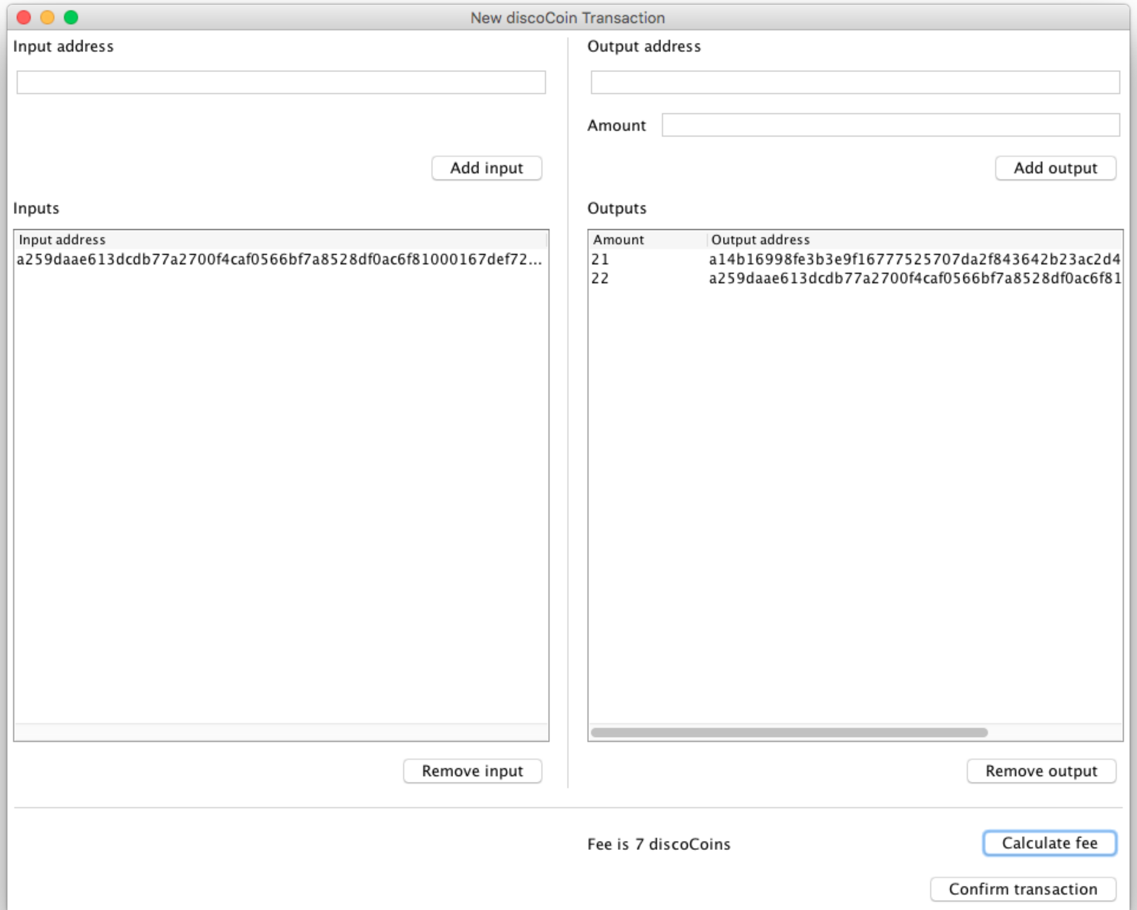

Figure 9: A `Transaction` consisting of one `Input` and two `Outputs`. An `Input` address must be accompanied with an `RSAPrivateKey` that is found in the wallet of the node. Behind the scenes, the `BlockExplorer` retrieves from the blockchain the value of the unspent `Output` referenced by the `Input`; in this case, the referenced `Output` has a value of 50 discōCoins resulting in a fee of 7 discōCoins.

A word on the security of storing `RSAPrivateKeys` in the `Inputs`. Only the node which creates the `Transaction` (i.e. the transactor) has access to the wallet that stores the `RSAPrivateKeys` belonging to those `Inputs`. When transmitting a `Transaction`, only the `TransactionReferences` and `Signatures` are included in the message to ensure that `RSAPrivateKeys` are not leaked.

6.2 NetworkNode

The main functions of the `NetworkNode` are to mine new blocks and to establish consensus. It has five threads that perform various functions: `SocketListener` keeps a server socket alive to receive incoming connection requests. `Pong` regularly broadcasts the `NetworkNode`'s height to its `Peers`. `BlockListener` continuously polls its `Peers` for any incoming messages and responds according to protocol. `ConsensusBuilder` periodically checks if any of its `Peers` have a longer blockchain and syncs to the `Peer` that has the longest chain. `Miner` continuously mines new blocks.

6.2.1 Creating a block

Step 1: Validating transactions

Before a `Transaction` can be included in a `Block`, it must be validated. The validation process involves two stages: (1) checking each `Input` against the latest list of unspent `Outputs`[7] (2) checking that there is a corresponding `Signature` that verifies against the `RSAPublicKey` of the `Output` that it references (*see* **verifying a signature**).

Algorithm 1: Validating a `Transaction` is a two-step process that checks that each `Input` references a valid unspent `Output` and verifies the corresponding `Signature`.

```
foreach transaction do
    foreach input do
        if valid unspent output then
            retrieve public key from unspent output;
            verify signature using public key;
            if valid signature then return true;
            else return false;
        else
            return false
        end
    end
end
```

[7] There is a subtle point to make about valid unspent `Outputs`. It is possible that more than one `Transaction` in the current pool of `Transactions` attempt to spend the same `Output`. Lines 998–1015 of `NetworkNode` check for these attempts. The first attempt to spend an `Output` is considered valid and all subsequent `Transactions` that attempt to spend the same `Output` are removed. However, it is a somewhat arbitrary decision as the order in which the `Transactions` are checked is not well-defined.

Step 2: Creating a mint transation

A mint `Transaction` is created and the fees from all the validated `Transactions` are added to its `Output`.

Monetary policy

The current minting protocol confers a miner's reward of 50 new discōCoins for each valid new `Block`. The amount of the reward is fixed hence the supply of discōCoins is infinite as long as new `Blocks` are being mined. The supply of fiat currencies such as the U.S. Dollar is theoretically infinite[8], although in practice, the supply is constrained by the U.S. Federal Reserve [24]. Conversely, the supply of Bitcoins is limited to 21 million [25], of which less than 1.5 million are currently in circulation[9]. Setting an appropriate monetary policy of discōCoin is itself a significant body of work to be addressed properly at a later date.

Step 3: Obtaining the Merkle root

The pool of `Transactions` (including the mint `Transaction`) is fed to the `MerkleTree` utility. Since a `MerkleTree` is a hashed binary tree, the number of leaves must be equal to a power of two e.g. 2, 4, 8, 16, and so on. If there are fewer `Transactions` than the minimum number of leaves required, the last `Transaction` is replicated as required.

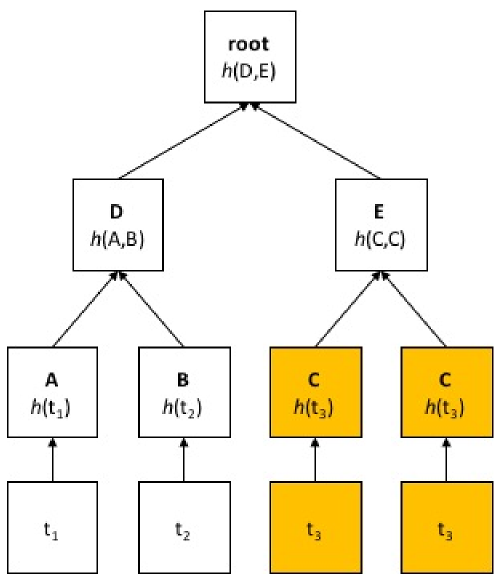

Figure 10: A `MerkleTree` containing three `Transactions` requires a minimum of four leaves hence t_3 is repeated once.

---

[8]The debt ceiling is an artificial construct that supposedly limits the supply of U.S. Dollars [23]; however, we make a distinction between arbitrary versus real limits, such as the finite supply of fossil fuels.

[9]According to `https://blockchain.info/stats` on September 11, 2016

Step 4: Finding a valid hash

Equation (6) is modified to include a nonce:

$$\text{block hash} = h(\text{previous block's hash} \mid \text{root of Merkle tree} \mid \text{nonce}) \tag{7}$$

As the root of the `MerkleTree` and the hash of the previous `Block` are both fixed, the nonce – which is just any number – functions like a dice roll causing the resulting hash to change in an unpredictable fashion (*see* **hiding**). The hash function is called repeatedly, each time incrementing the nonce[10], until a valid hash is found. The nonce that results in a valid hash is an important piece of information and is made public together with the resulting `Block`.

Difficulty threshold

The difficulty threshold for discōCoin was set based on the average time taken to mine ten blocks using one 2.3 GHz Intel Core i5 processor (*see* **hash rate test**).

Difficulty threshold	Time taken to mine ten blocks (miliseconds)	Average time taken to mine one block (minutes : seconds : miliseconds)
0ffffffff...f	676	0 : 0 : 68
00fffffff...f	736	0 : 0 : 74
000ffffff...f	919	0 : 0 : 92
0000fffff...f	1,661	0 : 0 : 166
00000ffff...f	43,735	0 : 4 : 374
000000fff...f	601,080	1 : 0 : 108
0000004fff...f	1,539,401	2 : 33 : 940
0000000fff...f	7,194,014	11 : 59 : 401

Figure 11: Summary of the results of the hash rate test.

Based the results, the difficulty threshold was set rather conservatively at 0000000fff...f to allow ample time for the `ConsensusBuilder` thread to initiate and complete the discōCoin consensus protocol.

[10] The nonce could be selected at random, but it is more efficient to systematically test nonce values given that the output of a hash function cannot be predicted from the input.

6.3 Communicating with peer nodes

In order for nodes to be able to understand each other, a messaging protocol is required. The messaging protocol is a set of rules about how to format (and parse) messages.

The `BlockListener` thread of the `NetworkNode` is responsible for continuously polling all the `Peers` that are connected to the `NetworkNode` to see if any messages have been received. It parses the message to determine the type of message and responds according to protocol.

Message id	Message type
0	Blockchain height broadcast
1	Request to verify the hash of a block at a given height
2	Response to a hash verification request
3	Request for block(s) by height
4	Response to block(s) by height request
5	Block broadcast
6	Transaction broadcast

Table 1: discōCoin messaging protocol.

6.3.1 Blockchain height broadcast

A blockchain height broadcast message is only sent by the `Pong` thread.

0	height of blockchain

Table 2: Message format for a blockchain height broadcast.

When a `NetworkNode` receives a height broadcast, it updates the known height of that `Peer`. The `ConsensusBuilder` thread of a `NetworkNode` will periodically check if the height of any of its `Peers` exceeds its own. The `ConsensusBuilder` will attempt to synchronise its own chain to the longest chain that it has knowledge of.

Algorithm 2: The algorithm for the `syncChain` method of the `ConsensusBuilder` thread.

```
1  foreach peer do
2      if height of peer > max height then
3          set max height = height of peer;
4          set chosen = peer;
5      end
6  end
7  set start = own height;
8  if max height > own height then
9      verify hash for block at height = start with chosen;
10     if hash fails to verify then
11         set start = start − 1;
12         go to line 9;
13     else
14         for i = start to height of chosen do
15             request for block at height = i from chosen;
16             if received block is valid then
17                 replace block at height = i with received block;
18             else
19                 go to line 15;
20             end
21         end
22     end
23 end
```

6.3.2 Request to verify the hash of a block at a given height

This message type is sent only from the `ConsensusBuilder` thread when it is attempting to synchronise its chain to the longest chain.

1	block height	block hash

Table 3: Message format for a request to verify the hash of a block at a given height.

6.3.3 Response to a hash verification request

When a `NetworkNode` receives a request to verify the hash of a block at a given height, it retrieves its own hash of the block at that height and compares it to the hash that was received with the request. It will respond with `true` if the received hash is the same as its own hash at that height, or `false`, otherwise.

2	block height	block hash	`true` or `false`

Table 4: Message format for a response to a hash verification request.

6.3.4 Request for block(s) by height

As with the request to verify the hash of a block at a given height, this message type is reserved for the `ConsensusBuilder` thread when it is synchronising its chain to the longest chain. The length of this message varies depending on the number of blocks the `ConsensusBuilder` is requesting for.

3	block height x	block height ...	block height y

Table 5: Message format for a request for block(s) by height.

6.3.5 Response to a block(s) by height request

The length of this message is dependent on the number of blocks requested.

4	block x	block ...	block y

Table 6: Message format for a response to a blocks(s) by height request.

Data serialisation

In order to be able to send complex data structures (e.g. `Blocks`) from one `NetworkNode` to another, the data structures need to be translated into a standardised format that can be transmitted and recreated on the other side.

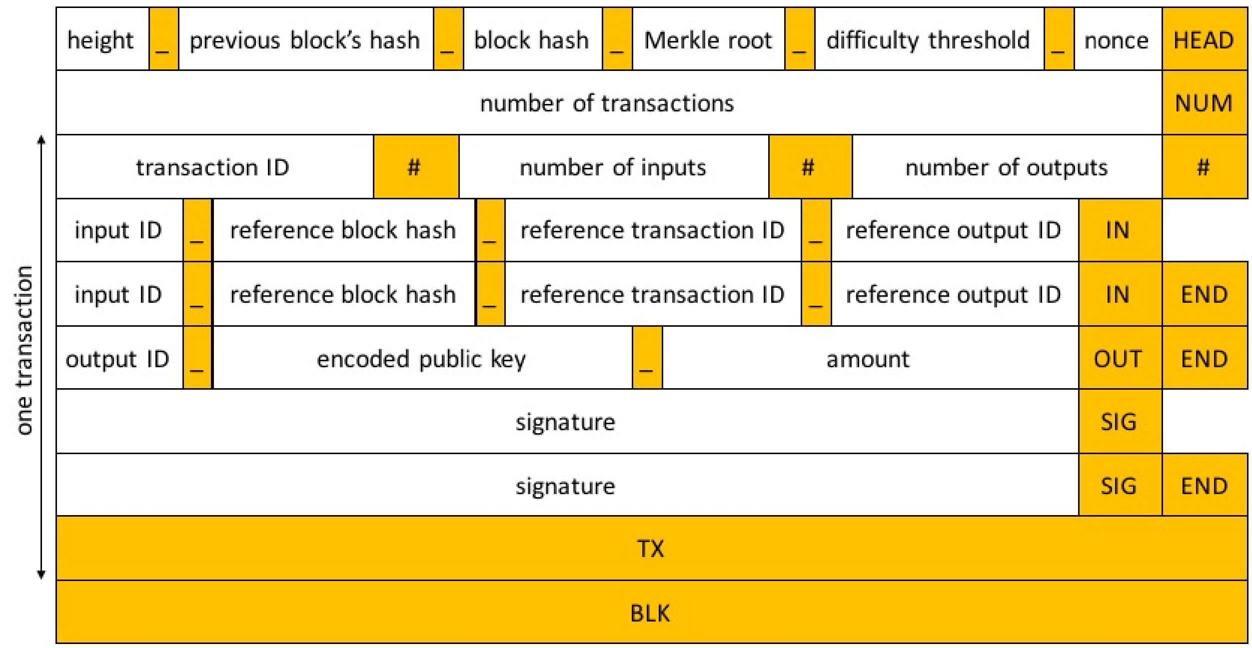

Figure 12: Serialised form of a discōCoin `Block` containing one `Transaction`. The `Transaction` consists of two `Inputs` (hence two `Signatures`) and one `Output`. The yellow boxes function as delimiters that separate the different pieces of information.

Validating a block

When a `NetworkNode` receives a `Block`, it goes through the following validation steps.

Step 1: Validating the block hash

In order to efficiently determine if a block hash is valid, the nonce must be provided. It is then possible to validate the hash in just one iteration of the hash function.

Step 2: Validating transactions

As detailed in the previous section.

Step 3: Validating the Merkle root

The last step in validating a block is to ensure that all `Transactions` that are included in the `Block` have been transmitted intact. This involves rebuilding the `MerkleTree` to see if the rebuilt root matches the given root.

6.3.6 Block broadcast

A block broadcast message is sent by a `NetworkNode` when it has mined a new `Block`. Upon receiving the broadcast and validating the `Block`, `NetworkNodes` will add the received `Block` to their blockchain.

5	block

Table 7: Message format for a block broadcast.

6.3.7 Transaction broadcast

After a new `Transaction` has been created and finalised, the `TransactionalNode` will connect to a `NetworkNode` in order to transmit its `Transaction`. After the `Transaction` has been validated by the `NetworkNode`, it will include it in the next `Block` that it attempts. The `NetworkNode` will also broadcast the `Transaction` to all its own `Peers`.

6	transaction

Table 8: Message format for a transaction broadcast.

6.4 Visualising the blockchain

Each `NetworkNode` maintains its own copy of the blockchain in XML format.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<blockchain>
  <block>
    <height>1</height>
    <header>
      <previousPoW>afd18fcd591cb7ae4e984fb91f94363293f0e82b611512b77f51218f26f367ec</previousPoW>
      <pow>000000031d7c3cdfe5a5fffab8fa158f931aafc1702980d5e98f4c7a2bae1895</pow>
      <merkleRoot>8e38684db696e58bf4cfb65ab20bd0b710c8940bfd54e5e2d3292bba8c2dec71</merkleRoot>
      <nonce>211435678</nonce>
      <difficulty>0000000fffffffffffffffffffffffffffffffffffffffffffffffffffffffff</difficulty>
    </header>
    <body>
      <transaction>
        <txID>1</txID>
        <output>
          <outputID>1</outputID>
          <outputAddress>50723e227f8395d46be5bcf8da39a6a0c08a319b1720dc0961b7d7707ac99d25</outputAddress>
          <publicKey>305c300d06...0203010001</publicKey>
          <amount>50</amount>
        </output>
      </transaction>
    </body>
  </block>
</blockchain>
```


Figure 13: Structure of the blockchain in XML showing the genesis block.

A simple website was built to provide a visual representation of the blockchain.

Figure 14: Website displaying the genesis block.

6.5 Results

Testing was performed by simulating nodes on the same computer. Nodes are able to create and broadcast transactions to be included in blocks, and are successful at cooperating to extend the blockchain by mining new blocks on top of each other.

6.6 Features planned for the next development phase

Given the speed of development, there are many features that remain unimplemented; please see the comments in the source code for more context.

The most pressing feature is the ability for `TransactionalNodes` to continually create new `Transactions`, which can easily be achieved through the addition of a "reset" button to the graphical user interface. Also, the ability in the graphical user interface to generate a new public key to be used as an output address would be very handy. Another user-friendly feature would be a function to update wallet balances based on the current list of unspent transaction outputs, accompanied by a graphical user interface for the wallet. A properly formatted log would be useful for debugging and testing purposes; being able to specify a verbose logging option at startup would also be a nice touch.

A more appropriate default port number for discōCoin should be selected after proper research; the current default port number is too low and may already be used by some other application. The robustness of the connection phase for a `NetworkNode` can be improved through better exception handling. There is also a resource leak from the `Scanner` attached to standard input; it may be possible to close and re-open it if the connection phase is triggered again.

A feedback loop should be put into place between the `Miner` and `BlockListener` threads such that the `Miner` knows to abandon mining a `Block` if a valid `Block` with the same height has been received by the `BlockListener`. At a later point, the discōCoin protocol should introduce the ability to reset the proof-of-work difficulty threshold based on the total hash power of the network.

The ability for a `NetworkNode` to request for a copy of all the `Transactions` that are in the memory pool of its `Peers` would become important as more `Transactions` are sent through the discōCoin network. Closely related is the devising of an efficient algorithm for identifying `Transactions` that can be safely discarded from the memory pool, such as `Transactions` which have already been included in a `Block` as well as invalid `Transactions`. Currently `NetworkNodes` retain all the `Transactions` they have received, but the retention does not result in those `Transactions` being replicated in the blockchain as `Transactions` which have been included in a previous `Block` would fail the unspent transaction output test.

At present, the `BlockListener` responds to a block(s) by height request in a single message. Breaking up the response such that there is only one block per message will likely result in a more efficient consensus process, as well as reduce the likelihood of errors in the event of a request for an unduly large number of blocks, such as when new `NetworkNodes` join the network in the future and request for the entire blockchain.

On the other end of the connection, if the `ConsensusBuilder` receives an invalid block during the chain synchronisation process, it should not keep adding blocks but instead send a fresh block(s) by height request. For safety, it should not remove any of its existing blocks until it has verified that the entire chain of blocks which it has been sent is valid as this leaves open a vector for attack. A malicious `Peer` could pretend to have a longer chain and inject an invalid block into the chain of blocks it sends in response.

Furthermore, the `ConsensusBuilder` thread goes into busy waiting twice during the consensus process, which is computationally expensive and wasteful. It would be interesting to properly implement semaphores instead.

Last but not least[11], hosting the website on a server would enable people to view discōCoin's blockchain without having to run a `NetworkNode` themselves.

7 Conclusion

Motivated by an interest in both cryptography and networking, the project set out to build a cryptocurrency from scratch. During the research phase, a second aim emerged: to produce a self-contained introduction to the technical underpinnings of cryptocurrencies that is accessible to readers who may not have a background in computer science.

Many popular cryptocurrencies, for example Namecoin and Litecoin, are forked from Bitcoin [26]. Building a cryptocurrency from scratch, even one as basic as discōCoin, presents a steep learning curve. As expected, the consensus layer was the most challenging aspect of the project. The approach was to start from a simple client-server implementation and gradually move towards a solution that works in a peer-to-peer model. The resulting protocol implementation reliably achieves consensus.

The project produced a cryptocurrency that fulfills the basic requirements of being able to transact in, and mine for, the cryptocurrency. Whether the secondary aim of the project has been met is left to the judgement of the reader. The experience has validated the author's research interests in the design of consensus protocols for distributed systems, and cryptocurrencies in general.

[11] Least but not last, proper removal of unspent transaction output nodes from the list so that no extraneous blank lines are written to file.

References

[1] S. Nakamoto. (2008). Bitcoin: A peer-to-peer electronic cash system, [Online]. Available: `https://bitcoin.org/bitcoin.pdf`.

[2] D. Boneh. (2015). Cryptography I, [Online]. Available: `https://www.coursera.org/learn/crypto`.

[3] A. Narayanan, J. Bonneau, E. Felten, A. Miller and J. Clark. (2015/2016). Bitcoin and cryptocurrency technologies, [Online]. Available: `https://www.coursera.org/course/bitcointech`.

[4] R. L. Rivest, A. Shamir and L. Adleman, 'A method for obtaining digital signatures and public-key cryptosystems', *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[5] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker and S. Savage, 'A fistful of Bitcoins: Characterizing payments among men with no names', in *Proceedings of the 2013 Conference on Internet Measurement Conference*, ACM, 2013, pp. 127–140.

[6] I. Miers, C. Garman, M. Green and A. D. Rubin, 'Zerocoin: Anonymous distributed e-cash from Bitcoin', in *2013 IEEE Symposium on Security and Privacy*, IEEE, 2013, pp. 397–411.

[7] R. C. Merkle, 'A digital signature based on a conventional encryption function', in *Conference on the Theory and Application of Cryptographic Techniques*, Springer, 1987, pp. 369–378.

[8] Various. (Nov. 2015). Genesis block, Bitcoin Wiki, [Online]. Available: `https://en.bitcoin.it/wiki/Genesis_block`.

[9] G. Wood, 'Ethereum: A secure decentralised generalised transaction ledger', *Ethereum Project Yellow Paper*, 2014. [Online]. Available: `http://gavwood.com/Paper.pdf`.

[10] M. Vasek, M. Thornton and T. Moore, 'Empirical analysis of denial-of-service attacks in the Bitcoin ecosystem', in *International Conference on Financial Cryptography and Data Security*, Springer, 2014, pp. 57–71.

[11] B. Dickerson. (2005). Notes on computer networks, [Online]. Available: `http://homepages.herts.ac.uk/~comqrgd/docs/network-notes/network-notes.pdf`.

[12] B. Eater. (Oct. 2014). Networking tutorial, [Online]. Available: https://www.youtube.com/user/eaterbc/playlists.

[13] T. Kantor. (2015). Ulterior states, [Online]. Available: http://www.iamsatoshi.com/.

[14] A. Back. (2002). Hashcash: A denial-of-service counter-measure, [Online]. Available: http://www.hashcash.org/papers/hashcash.pdf.

[15] C. Dwork and M. Naor, 'Pricing via processing or combatting junk mail', in *Annual International Cryptology Conference*, Springer, 1992, pp. 139–147.

[16] QuantumMechanic. (Jul. 2011). Proof-of-stake instead of proof-of-work, Bitcoin Forum, [Online]. Available: https://bitcointalk.org/index.php?topic=27787.0.

[17] I. Bentov, C. Lee, A. Mizrahi and M. Rosenfeld, 'Proof-of-activity: Extending Bitcoin's proof-of-work via proof-of-stake', *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 3, pp. 34–37, 2014.

[18] V. Buterin. (Jul. 2014). On stake, [Online]. Available: https://blog.%20ethereum.%20org/2014/07/05/stake.

[19] S. Park, K. Pietrzak, J. Alwen, G. Fuchsbauer and P. Gazi, 'Spacecoin: A cryptocurrency based on proofs of space', IACR Cryptology ePrint Archive, 2015: 528, Tech. Rep., 2015.

[20] L. Lamport, 'Paxos made simple', *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[21] W. Zhao, 'A Byzantine fault-tolerant distributed-commit protocol', in *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*, IEEE, 2007, pp. 37–46.

[22] L. Lamport, R. Shostak and M. Pease, 'The Byzantine generals problem', *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[23] (Aug. 2016). Debt limit, U.S. Department of the Treasury, [Online]. Available: https://www.treasury.gov/initiatives/Pages/debtlimit.aspx.

[24] (Feb. 2014). What is the purpose of the Federal Reserve System?, Board of Governors of the Federal Reserve System, [Online]. Available: http://www.federalreserve.gov/faqs/about_12594.htm.

[25] Various. (Jul. 2016). Controlled supply, Bitcoin Wiki, [Online]. Available: https://en.bitcoin.it/wiki/Controlled_supply.

[26] ——, (Sep. 2014). Forks, Bitcoin Wiki, [Online]. Available: http://en.bitcoinwiki.org/Forks.

[27] R. C. Merkle, 'Protocols for public key cryptosystems', in *IEEE Symposium on Security and Privacy*, vol. 122, 1980.

[28] H. Massias, X. S. Avila and J.-J. Quisquater, 'Design of a secure timestamping service with minimal trust requirement', in *The 20th Symposium on Information Theory in the Benelux*, 1999.

[29] R. Patterson. (Aug. 2015). Alternatives for proof-of-work, Part 1: Proof-of-stake, Byte-Coin, [Online]. Available: https://bytecoin.org/blog/proof-of-stake-proof-of-work-comparison/.

[30] A. Swartz, *Squaring the triangle: Secure, decentralized, human-readable names*, 2011. [Online]. Available: http://www.aaronsw.com/weblog/squarezooko.

[31] Various. (Aug. 2016). Mining, Bitcoin Wiki, [Online]. Available: https://en.bitcoin.it/wiki/Mining.

[32] ——, (May 2016). Bitcoin protocol documentation, Bitcoin Wiki, [Online]. Available: https://en.bitcoin.it/wiki/Protocol_documentation.

[33] ——, (Mar. 2016). Bitcoin protocol rules, Bitcoin Wiki, [Online]. Available: https://en.bitcoin.it/wiki/Protocol_rules.

[34] ——, (Feb. 2016). Bitcoin network, Bitcoin Wiki, [Online]. Available: https://en.bitcoin.it/wiki/Network.

[35] ——, (Mar. 2014). Satoshi client node discovery, Bitcoin Wiki, [Online]. Available: https://en.bitcoin.it/wiki/Satoshi_Client_Node_Discovery.

[36] A. Chepurnoy. (Feb. 2016). The architecture of a cryptocurrency, Input Output HK, [Online]. Available: https://github.com/input-output-hk/Scorex/blob/master/docs/articles/components.md.

8 Appendices

8.1 Project management

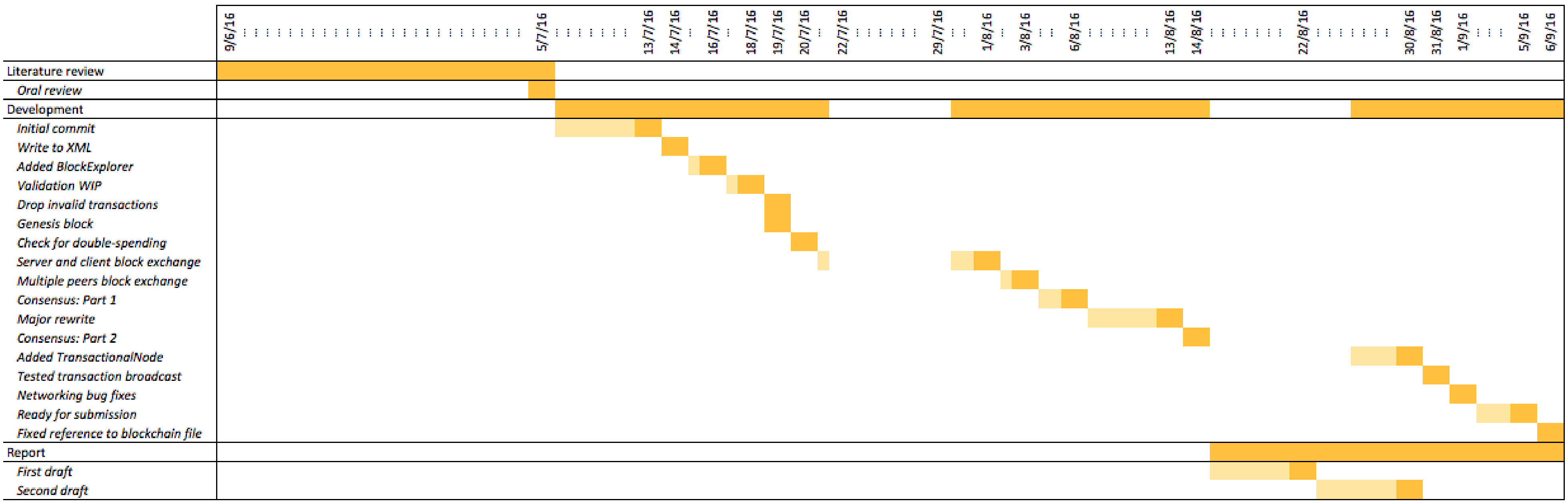

Figure 15: Project Gantt chart. Darker yellow signifies milestones e.g. code commits.

8.2 User manual

1. **Unziping the archive**

   Ensure that the archive is located in the desired directory. Navigate to that directory and run the following command from the terminal:

   ```
   $ unzip discoCoin.zip
   ```

2. **Compiling the source code**

   Make a separate directory for the binaries, then run the following command to compile the source code:

   ```
   $ javac -d [directory] *.java
   ```

3. **Launching the application**

   There are three ways to launch the application. The following command will launch a pure miner node:

   ```
   $ java p2p/NetworkNode [port]
   ```

   The default port is 5003. If there are no active nodes on 5003, nodes running on other ports will not be able to request the genesis block. There is no graphical interface for miner nodes.

   There is an option to launch a node that has both mining and transacting capabilities by invoking the `-t` flag:

   ```
   $ java p2p/NetworkNode -t [port]
   ```

   Lastly, it is possible to launch a pure transactor node. However, the hostname and port number of an active miner node must be provided as arguments.

   ```
   $ java gui/TransactionalNode [hostname] [port]
   ```

   Note that there needs to be at least two active `NetworkNodes` that are connected to each other in order for the consensus protocol to be deployed.

Sample console output upon launching the application:

```
2016-09-02T18:58:15.354 I am 0.0.0.0 5003
2016-09-02T18:58:15.446 SocketListener thread started
Connect to peer? (Y/N): y
Hostname: 0.0.0.0
Port: 5004
Connect to peer? (Y/N): n
2016-09-02T18:58:17.178 BlockListener thread started
2016-09-02T18:58:17.181 Pong thread started
2016-09-02T18:58:17.185 Own chain height: 2
2016-09-02T18:58:17.186 Max chain height: 0
2016-09-02T18:58:17.187 Miner thread started
2016-09-02T18:58:17.188 ConsensusBuilder thread started
```

Note that if Alice wants to connect to Bob, she must wait until Bob's `SocketListener` thread has started, and vice versa.

4. **Making a transaction**

   Launch an instance of the `TransactionalNode` and click on the "New Transaction" button.

   In the "Input address" field, copy and paste a unspent transaction output address from the blockchain. Note that the address and corresponding private key must be found in the wallet of the transacting node. To add the `Input`, click on the "Add input" button. To add an `Output`, copy and paste any valid output address, enter an amount and click "Add output".

   For convenience, the following values may be used with the .zip file that was submitted along with this report:

   Input address

   50723e227f8395d46be5bcf8da39a6a0c08a319b1720dc0961b7d7707ac99d25

   Output address

   a14b16998fe3b3e9f16777525707da2f843642b23ac2d40c195456b88c23a740

A configuration that works is to launch one instance of the application using $ java p2p/NetworkNode −t 5003 and another instance using, for example, $ java p2p/NetworkNode 5004. The private key for the input address given here is stored in dat/wallet_0 .0.0.0 _5003.xml, hence, to send a `Transaction`, please do so from the instance running on port 5003.

Please note that there is a lag between a `NetworkNode` receiving a `Transaction` and incorporating it into a `Block`. This is because the `NetworkNode` will continue to mine the `Block` that it was working on when it received the `Transaction`, therefore the new `Transaction` will only be incorporated into the next `Block` that it attempts.

5. **Viewing the blockchain**
   The website is not hosted on a server hence it only displays using a Safari web browser. To watch new blocks being mined, ensure that an instance of `NetworkNode` is running on port 5003 and run the following command:

   ```
   $ open −a Safari www/index.html
   ```

   The website will automatically refresh its data every minute.

8.3 Hash rate test

```java
package test;

import java.io.IOException;
import java.time.LocalDateTime;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import p2p.NetworkNode;
import util.Filename;

public class HashRateTest {

  public static void main (String[] args)
    throws SAXException, IOException, ParserConfigurationException {

    NetworkNode node = new NetworkNode();

    String directory = "dat/";
    String extension = ".xml";

    Filename blockchainFilename = new Filename(directory, "blockchain", extension);
    Filename utxoFilename = new Filename(directory, "utxolist", extension);
    Filename walletFilename = new Filename(directory, "wallet", extension);

    node.initialiseExplorers(blockchainFilename, utxoFilename, walletFilename);
    System.out.println(LocalDateTime.now() + " " + node.difficulty);

    for (int i = 0; i < 10; i++) node.mine();
  }
}
```

```
2016-09-02T20:42:12.513 0fffffffff...f
2016-09-02T20:42:12.772 Found nonce 6 and hash 0b7f...
2016-09-02T20:42:12.775 Added new block 0b7f...
2016-09-02T20:42:12.843 Found nonce 5 and hash 08c3...
2016-09-02T20:42:12.845 Added new block 08c3...
2016-09-02T20:42:12.888 Found nonce 1 and hash 06fc...
2016-09-02T20:42:12.888 Added new block 06fc...
2016-09-02T20:42:12.940 Found nonce 21 and hash 0559...
2016-09-02T20:42:12.941 Added new block 0559...
2016-09-02T20:42:12.972 Found nonce 10 and hash 0c50...
2016-09-02T20:42:12.973 Added new block 0c50...
2016-09-02T20:42:12.997 Found nonce 26 and hash 0356...
2016-09-02T20:42:12.998 Added new block 0356...
2016-09-02T20:42:13.055 Found nonce 7 and hash 020a...
2016-09-02T20:42:13.056 Added new block 020a...
2016-09-02T20:42:13.099 Found nonce 9 and hash 000f...
2016-09-02T20:42:13.103 Added new block 000f...
2016-09-02T20:42:13.133 Found nonce 0 and hash 0692...
2016-09-02T20:42:13.134 Added new block 0692...
2016-09-02T20:42:13.188 Found nonce 26 and hash 09d9...
2016-09-02T20:42:13.189 Added new block 09d9...
```

```
2016-09-02T20:41:52.713 00ffffffff...f
2016-09-02T20:41:52.955 Found nonce 95 and hash 00e65...
2016-09-02T20:41:52.957 Added new block 00e65...
2016-09-02T20:41:53.045 Found nonce 100 and hash 00fc4...
2016-09-02T20:41:53.046 Added new block 00fc4...
2016-09-02T20:41:53.112 Found nonce 277 and hash 0059a...
2016-09-02T20:41:53.113 Added new block 0059a...
2016-09-02T20:41:53.193 Found nonce 553 and hash 00089...
2016-09-02T20:41:53.194 Added new block 00089...
2016-09-02T20:41:53.254 Found nonce 134 and hash 007c1...
2016-09-02T20:41:53.255 Added new block 007c1...
2016-09-02T20:41:53.303 Found nonce 154 and hash 00e5b...
2016-09-02T20:41:53.304 Added new block 00e5b...
2016-09-02T20:41:53.341 Found nonce 448 and hash 0074a...
2016-09-02T20:41:53.342 Added new block 0074a...
2016-09-02T20:41:53.378 Found nonce 19 and hash 00e5c...
2016-09-02T20:41:53.379 Added new block 00e5c...
2016-09-02T20:41:53.415 Found nonce 217 and hash 00d2f...
2016-09-02T20:41:53.416 Added new block 00d2f...
2016-09-02T20:41:53.448 Found nonce 81 and hash 00ada...
2016-09-02T20:41:53.449 Added new block 00ada...
```

```
2016-09-02T20:41:36.250 000fffffff...f
2016-09-02T20:41:36.575 Found nonce 2926 and hash 000026...
2016-09-02T20:41:36.576 Added new block 000026...
2016-09-02T20:41:36.690 Found nonce 4949 and hash 000485...
2016-09-02T20:41:36.691 Added new block 000485...
2016-09-02T20:41:36.771 Found nonce 4829 and hash 0000ad...
2016-09-02T20:41:36.772 Added new block 0000ad...
2016-09-02T20:41:36.881 Found nonce 8560 and hash 000561...
2016-09-02T20:41:36.881 Added new block 000561...
2016-09-02T20:41:36.912 Found nonce 362 and hash 000890...
2016-09-02T20:41:36.913 Added new block 000890...
2016-09-02T20:41:36.975 Found nonce 4218 and hash 00008d...
2016-09-02T20:41:36.977 Added new block 00008d...
2016-09-02T20:41:37.018 Found nonce 2996 and hash 0002e1...
2016-09-02T20:41:37.019 Added new block 0002e1...
2016-09-02T20:41:37.086 Found nonce 2113 and hash 000380...
2016-09-02T20:41:37.087 Added new block 000380...
2016-09-02T20:41:37.131 Found nonce 1668 and hash 0007ba...
2016-09-02T20:41:37.131 Added new block 0007ba...
2016-09-02T20:41:37.168 Found nonce 1424 and hash 00026b...
2016-09-02T20:41:37.169 Added new block 00026b...
```

```
2016-09-02T20:40:54.561 0000ffffff...f
2016-09-02T20:40:55.020 Found nonce 13004 and hash 0000be8...
2016-09-02T20:40:55.023 Added new block 0000be8...
2016-09-02T20:40:55.205 Found nonce 13397 and hash 0000672...
2016-09-02T20:40:55.206 Added new block 0000672...
2016-09-02T20:40:55.338 Found nonce 16576 and hash 0000ed6...
2016-09-02T20:40:55.339 Added new block 0000ed6...
2016-09-02T20:40:55.432 Found nonce 10782 and hash 0000fe3...
2016-09-02T20:40:55.433 Added new block 0000fe3...
2016-09-02T20:40:55.533 Found nonce 12300 and hash 0000896...
2016-09-02T20:40:55.534 Added new block 0000896...
2016-09-02T20:40:55.666 Found nonce 23762 and hash 0000437...
2016-09-02T20:40:55.667 Added new block 0000437...
2016-09-02T20:40:55.929 Found nonce 34440 and hash 000064d...
2016-09-02T20:40:55.929 Added new block 000064d...
2016-09-02T20:40:56.044 Found nonce 19321 and hash 0000112...
2016-09-02T20:40:56.044 Added new block 0000112...
2016-09-02T20:40:56.125 Found nonce 12565 and hash 0000ed8...
2016-09-02T20:40:56.126 Added new block 0000ed8...
2016-09-02T20:40:56.221 Found nonce 16068 and hash 00004e6...
2016-09-02T20:40:56.222 Added new block 00004e6...
```

```
2016-09-02T20:36:56.780 00000fffff...f
2016-09-02T20:37:01.502 Found nonce 900893 and hash 00000c2f...
2016-09-02T20:37:01.506 Added new block 00000c2f...
2016-09-02T20:37:05.139 Found nonce 866058 and hash 00000800...
2016-09-02T20:37:05.140 Added new block 00000800...
2016-09-02T20:37:05.684 Found nonce 120615 and hash 000008bf...
2016-09-02T20:37:05.685 Added new block 000008bf...
2016-09-02T20:37:16.345 Found nonce 2630682 and hash 00000c61...
2016-09-02T20:37:16.346 Added new block 00000c61...
2016-09-02T20:37:16.711 Found nonce 42819 and hash 000007a5...
2016-09-02T20:37:16.712 Added new block 000007a5...
2016-09-02T20:37:19.313 Found nonce 625482 and hash 00000672...
2016-09-02T20:37:19.313 Added new block 00000672...
2016-09-02T20:37:20.142 Found nonce 186512 and hash 00000b58...
2016-09-02T20:37:20.143 Added new block 00000b58...
2016-09-02T20:37:29.440 Found nonce 2233925 and hash 000001e0...
2016-09-02T20:37:29.440 Added new block 000001e0...
2016-09-02T20:37:33.347 Found nonce 933106 and hash 000004cb...
2016-09-02T20:37:33.347 Added new block 000004cb...
2016-09-02T20:37:40.515 Found nonce 1794896 and hash 00000659...
2016-09-02T20:37:40.515 Added new block 00000659...
```

```
2016-09-02T20:11:18.114 000000ffff...f
2016-09-02T20:11:24.166 Found nonce 1332308 and hash 000000e00...
2016-09-02T20:11:24.201 Added new block 000000e00...
2016-09-02T20:12:26.992 Found nonce 15505742 and hash 000000e5f...
2016-09-02T20:12:26.993 Added new block 000000e5f...
2016-09-02T20:13:11.777 Found nonce 11081416 and hash 000000a7f...
2016-09-02T20:13:11.778 Added new block 000000a7f...
2016-09-02T20:17:40.118 Found nonce 65602150 and hash 0000006ed...
2016-09-02T20:17:40.119 Added new block 0000006ed...
2016-09-02T20:18:15.907 Found nonce 8596283 and hash 000000d19...
2016-09-02T20:18:15.907 Added new block 000000d19...
2016-09-02T20:18:48.240 Found nonce 7842623 and hash 000000c19...
2016-09-02T20:18:48.241 Added new block 000000c19...
2016-09-02T20:19:35.708 Found nonce 11663763 and hash 00000060f...
2016-09-02T20:19:35.709 Added new block 00000060f...
2016-09-02T20:19:59.971 Found nonce 5990690 and hash 000000d23...
2016-09-02T20:19:59.971 Added new block 000000d23...
2016-09-02T20:21:08.256 Found nonce 16887466 and hash 000000d53...
2016-09-02T20:21:08.257 Added new block 000000d53...
2016-09-02T20:21:19.194 Found nonce 2696912 and hash 000000423...
2016-09-02T20:21:19.194 Added new block 000000423...
```

```
2016-09-02T21:29:01.716  0000004fff...f
2016-09-02T21:29:15.030  Found nonce 2994539 and hash 000000380e...
2016-09-02T21:29:15.040  Added new block 000000380e...
2016-09-02T21:34:05.068  Found nonce 70576826 and hash 0000003194...
2016-09-02T21:34:05.069  Added new block 0000003194...
2016-09-02T21:40:29.455  Found nonce 93065881 and hash 0000000fcf...
2016-09-02T21:40:29.457  Added new block 0000000fcf...
2016-09-02T21:40:32.189  Found nonce 641968 and hash 0000003523...
2016-09-02T21:40:32.190  Added new block 0000003523...
2016-09-02T21:44:03.564  Found nonce 51364131 and hash 00000010b2...
2016-09-02T21:44:03.565  Added new block 00000010b2...
2016-09-02T21:45:47.657  Found nonce 25360499 and hash 000000063e...
2016-09-02T21:45:47.657  Added new block 000000063e...
2016-09-02T21:51:54.632  Found nonce 89196498 and hash 0000000d38...
2016-09-02T21:51:54.633  Added new block 0000000d38...
2016-09-02T21:53:11.452  Found nonce 18663967 and hash 00000032e8...
2016-09-02T21:53:11.453  Added new block 00000032e8...
2016-09-02T21:53:24.065  Found nonce 3036342 and hash 0000004ff4...
2016-09-02T21:53:24.066  Added new block 0000004ff4...
2016-09-02T21:54:41.117  Found nonce 18684953 and hash 000000170b...
2016-09-02T21:54:41.117  Added new block 000000170b...
```

```
2016-09-02T22:47:06.635  0000000fff...f
2016-09-02T22:48:57.661  Found nonce 27289140 and hash 0000000ad5...
2016-09-02T22:48:57.758  Added new block 0000000ad5...
2016-09-02T22:49:25.286  Found nonce 6595805 and hash 0000000c92...
2016-09-02T22:49:25.287  Added new block 0000000c92...
2016-09-02T22:50:06.779  Found nonce 10178045 and hash 0000000c05...
2016-09-02T22:50:06.780  Added new block 0000000c05...
2016-09-02T22:56:58.844  Found nonce 103125444 and hash 0000000fb3...
2016-09-02T22:56:58.845  Added new block 0000000fb3...
2016-09-02T23:13:54.852  Found nonce 48753887 and hash 0000000c1e...
2016-09-02T23:13:54.854  Added new block 0000000c1e...
2016-09-02T23:27:11.455  Found nonce 197328361 and hash 0000000e64...
2016-09-02T23:27:11.463  Added new block 0000000e64...
2016-09-02T23:51:16.441  Found nonce 333395837 and hash 0000000a95...
2016-09-02T23:51:16.442  Added new block 0000000a95...
2016-09-03T00:23:58.478  Found nonce 467707027 and hash 00000002e2...
2016-09-03T00:23:58.483  Added new block 00000002e2...
2016-09-03T00:43:38.109  Found nonce 272200517 and hash 0000000067...
2016-09-03T00:43:38.114  Added new block 0000000067...
2016-09-03T00:47:00.648  Found nonce 46474673 and hash 00000006c4...
2016-09-03T00:47:00.649  Added new block 00000006c4...
```

8.4 Source code

The full source code is available at `https://github.com/yinyee/coin`.

8.4.1 Directory structure

```
src
├── obj
│   ├── Input.java
│   ├── Output.java
│   ├── Transaction.java
│   ├── TransactionReference.java
│   ├── Block.java
│   └── BlockHeader.java
├── p2p
│   ├── NetworkNode.java
│   └── Peer.java
├── gui
│   ├── TransactionalNode.java
│   └── InputsOutputs.java
├── util
│   ├── SHA256.java
│   ├── RSA512.java
│   ├── Signature.java
│   ├── MerkleTree.java
│   ├── BaseConverter.java
│   ├── Filename.java
│   ├── BlockExplorer.java
│   ├── UTXOExplorer.java
│   ├── WalletExplorer.java
│   └── XMLio.java
├── test
│   ├── GenesisBlock.java
│   ├── HashRateTest.java
│   ├── CompareChainsTest.java
│   ├── SignatureRunTest.java
│   └── WalletRunTest.java
├── dat
│   ├── blockchain_0.0.0.0_5003.xml
│   ├── utxolist_0.0.0.0_5003.xml
│   └── wallet_0.0.0.0_5003.xml
└── www
    ├── index.html
    ├── scripts.js
    └── styles.css
```

8.4.2 NetworkNode.java

The various parts of the class are presented in logical groups. This differs from the actual code where, for ease of de-bugging, methods are grouped with the threads in which they are most often used.

Package declaration

```java
package p2p;
```

Imports from standard libraries

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.NoSuchAlgorithmException;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.X509EncodedKeySpec;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.Scanner;

import org.w3c.dom.Node;
```

Imports from other packages

```java
import gui.TransactionalNode;
import obj.Block;
import obj.BlockHeader;
import obj.BlockHeader.BlockHeaderException;
import obj.Input;
import obj.Output;
import obj.Transaction;
import obj.TransactionReference;
import obj.Transaction.TransactionException;
import util.BaseConverter;
import util.BlockExplorer;
import util.Filename;
import util.MerkleTree;
import util.MerkleTree.MerkleTreeException;
import util.RSA512;
import util.SHA256;
import util.UTXOExplorer;
import util.WalletExplorer;
```

Declarations

```java
public class NetworkNode {

  // File information
  public final static String DIR = "dat/";
  public final static String BLOCKCHAIN = "blockchain";
  public final static String UTXO = "utxolist";
  public final static String WALLET = "wallet";
  public final static String EXT = ".xml";

  public final static int REWARD = 50;
  public String difficulty = "0000000fffffffffffffffffffffffffffffffffffffffffffffffffffffffff";

  private String hostname;
  private int port;
  private ServerSocket server;

  private ArrayList<Peer> peers;
  private ArrayList<PrintWriter> toPeers;
  private String msgSeparator = "MSG";
  private String blockSeparator = "BLK";
  private String separator = " ";

  private BlockExplorer blockExplorer;
  private UTXOExplorer utxoExplorer;
  private WalletExplorer walletExplorer;

  private ArrayList<Transaction> mempool;

  private ConsensusBuilder consensusBuilder;
  private Miner miner;
```

Constructor

```java
public NetworkNode() {
  peers = new ArrayList<Peer>();
  toPeers = new ArrayList<PrintWriter>();
  mempool = new ArrayList<Transaction>();
}
```

Main method

```java
public static void main (String[] args) {

  NetworkNode node = new NetworkNode();

  // Parse arguments
  if (args[0].compareTo("-t") == 0) /*Launch TransactionalNode*/ {

    if (args.length != 2) {
      System.err.println("Arguments: [-t (optional)] [own port]");
      System.exit(0);
    } else {

      // Start server
      node.port = Integer.valueOf(args[1]);
      try {
        node.server = new ServerSocket(node.port);
      } catch (IOException e) {
        e.printStackTrace();
      }
      node.hostname = node.server.getInetAddress().getHostAddress();

      // Launch TransactionalNode
      TransactionalNode.getInstance(node.hostname, node.port);
    }
  }

```

```java
    else /*Do not launch TransactionalNode*/ {

      if (args.length != 1) {
        System.err.println("Arguments: [-t (optional)] [own port]");
        System.exit(0);
      } else {

        // Start server
        node.port = Integer.valueOf(args[0]);
        try {
          node.server = new ServerSocket(node.port);
        } catch (IOException e) {
          e.printStackTrace();
        }
        node.hostname = node.server.getInetAddress().getHostAddress();
      }
    }

    System.out.println(LocalDateTime.now() + " I am " + node.hostname + " " + node.port);

    // Keep listening for connection requests
    SocketListener socketListener = node.new SocketListener(node.server);
    socketListener.start();

    // Initialize explorers
    String myHostname = node.hostname;
    String myPort = String.valueOf(node.port);
    Filename blockchainFile = new Filename(DIR, BLOCKCHAIN, EXT, myHostname, myPort);
    Filename utxoFile = new Filename(DIR, UTXO, EXT, myHostname, myPort);
    Filename walletFile = new Filename(DIR, WALLET, EXT, myHostname, myPort);

    node.initialiseExplorers(blockchainFile, utxoFile, walletFile);
    node.utxoExplorer.rebuildUTXOList(node.blockExplorer);


```

```java
    // Connect to peers
    node.findPeers();

    // Start listening to peers
    BlockListener listener = node.new BlockListener();
    listener.start();

    // Start broadcasting height
    Pong pong = node.new Pong();
    pong.start();

    // Start syncing chain
    node.consensusBuilder = node.new ConsensusBuilder();
    node.consensusBuilder.syncChain();

    // Start mining
    node.miner = node.new Miner();
    node.miner.start();

    node.consensusBuilder.start();
```

SocketListener thread

```java
/**
 * Keeps a server socket live to accept any incoming connection requests.
 */
class SocketListener extends Thread {

  private ServerSocket server;

  public SocketListener(ServerSocket server) {
    super("Socket listener");
    this.server = server;
  }

  public void run() {
    System.out.println(LocalDateTime.now() + " SocketListener thread started");
    while (true) {
      if (this.isInterrupted()) System.out.println(LocalDateTime.now() + " SocketListener is interrupted");
      try {
        Socket socket = server.accept();
        connect(socket);
        System.out.println(LocalDateTime.now() + " Connection received from " +
          socket.getInetAddress().getHostName() + " " + socket.getPort());
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

BlockListener thread

```
/**
 * Continuously polls each peer for any incoming messages and responds accordingly to different message types.
 */
class BlockListener extends Thread {

  public BlockListener() {
    super("Block listener");
  }

  public void run() {

    System.out.println(LocalDateTime.now() + " BlockListener thread started");

    BufferedReader reader;
    Peer peer;
    String message, msgType, msgBody, responseMsgType, response;
    String[] split, s;
    String height, pow, same;
    int iHeight;

    while (true) {

      if (this.isInterrupted()) System.out.println(LocalDateTime.now() + " BlockListener is interrupted");

      for (int i = 0; i < peers.size(); i++) {

        peer = null;
        message = null;
        msgType = null;
        msgBody = null;

        if (peers.get(i) != null) {
          peer = peers.get(i);
          reader = peer.reader();
```

```java
            try {
              message = reader.readLine();
            } catch (Exception e) {
              System.out.println(LocalDateTime.now() + " Removed " + peer.hostname()+ " " + peer.port());
              peers.remove(i);
              toPeers.remove(i);
              e.printStackTrace();
            }
          }

          if (message != null) {

            split = message.split(msgSeparator);
            msgType = split[0];
            if (split.length > 1) msgBody = split[1];

            switch (msgType) {

              case "0": /*Height broadcast*/ {
                int peerHeight = Integer.valueOf(msgBody);
                peer.updateCurrentHeight(peerHeight);
                break;
              }

              case "1": /*Check hash by height*/ {
                s = msgBody.split(separator);
                height = s[0];
                pow = s[1];

                responseMsgType = "2";
                response = height + separator + pow + separator;

                if (pow.compareTo(blockExplorer.getPoWByHeight(height)) == 0) response += "true";
                else response += "false";
                peer.writer().println(responseMsgType + msgSeparator + response);
```

```java
                System.out.println(LocalDateTime.now() + " Sent: " + responseMsgType + msgSeparator + response);
                break;
            }

            case "2": /*Check hash by height response*/ {

                s = msgBody.split(separator);
                height = s[0];
                iHeight = Integer.valueOf(height);
                pow = s[1];
                same = s[2];

                // If hash at height is correct, set ConsensusBuilder's max consensus height
                if (same.compareTo("true") == 0) {
                    consensusBuilder.setMaxConsensusHeight(iHeight);
                    break;
                }

                // If hash at height is incorrect, check hash at height - 1
                if (same.compareTo("false") == 0) {
                    responseMsgType = "1";
                    String requestHeight = String.valueOf(iHeight - 1);
                    response = requestHeight + separator + blockExplorer.getPoWByHeight(requestHeight);
                    peer.writer().println(responseMsgType + msgSeparator + response);
                    System.out.println(LocalDateTime.now() + " Sent: " + responseMsgType + msgSeparator + response);
                    break;
                }

                break;
            }

            case "3": /*Block by height request*/ {

                responseMsgType = "4";
                response = new String();
```

```

        s = msgBody.split(separator);

        for (int j = 0; j < s.length; j++)
          response += blockExplorer.getBlock(s[j]).toString() + blockSeparator;

        peer.writer().println(responseMsgType + msgSeparator + response);
        System.out.println(LocalDateTime.now() + " Sent: " + responseMsgType + msgSeparator + response);
        break;
      }

      // TODO Consider breaking up into a series of messages instead
      case "4": /*Block by height response */ {

        s = msgBody.split(blockSeparator);

        Block block;
        for (int j = 0; j < s.length; j++) {
          try {
            block = readBlock(s[j]);
            peer.storedBlocks().add(block);
          } catch (Exception e) {
            e.printStackTrace();
          }
        }
        break;
      }

      case "5": /*Block broadcast*/ {

        System.out.println(LocalDateTime.now() + " Receiving block from " +
          peer.hostname() + " port " + peer.port());

        Block block = null;
        boolean valid = false;
```

```
        try {
          block = readBlock(msgBody);
          valid = block.validate(blockExplorer, utxoExplorer);
        } catch (Exception e) {
          e.printStackTrace();
        }

        if (valid & isNewBlock(block)) {
          blockExplorer.addNewBlock(block);
          blockExplorer.updateBlockchainFile();
          utxoExplorer.rebuildUTXOList(blockExplorer);
          broadcast("5", msgBody);
        }

        break;
      }

      case "6": /*Transaction broadcast*/ {

        System.out.println(LocalDateTime.now() + " Receiving transaction from " +
          peer.hostname() + " port " + peer.port());

        String[] parts = msgBody.split("TX");

        Transaction transaction = null;
        boolean valid = false;

        try {
          transaction = readTransaction(parts[0]);
          valid = transaction.validate(blockExplorer, utxoExplorer);
        } catch (TransactionException txe) {
          /*Input not found in UTXO list because blocks have not been added yet*/;
        } catch (Exception e) {
          e.printStackTrace();
```

```
                }

                if (valid & isNewTransaction(transaction)) {
                  mempool.add(transaction);
                  System.out.println(LocalDateTime.now() +
                    " Transaction validated and added to the memory pool.
                    Number of transactions in memory pool: " + mempool.size());
                  peer.writer().println("OK"); // TODO
                  broadcast("6", msgBody);
                } else peer.writer().println("FAIL");

                break;
              }
            }
          }
        }
      }
    }
  }
```

ConsensusBuilder thread

```java
/**
 * Periodically checks for the longest chain and syncs to it.
 */
class ConsensusBuilder extends Thread {

  private int maxConsensusHeight;
  private int numberOfBlocks;

  public ConsensusBuilder() {
    super("Consensus builder");
  }

  public void run() {

    System.out.println(LocalDateTime.now() + " ConsensusBuilder thread started");

    while (true) {
      if (this.isInterrupted()) System.out.println(LocalDateTime.now() + " ConsensusBuilder is interrupted");
      syncChain();
      try {
        sleep(60000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  private void setMaxConsensusHeight(int maxConsensusHeight) {
    this.maxConsensusHeight = maxConsensusHeight;
  }

  private void syncChain() {

    Peer peer, syncToPeer = null;
```

```java
    int ownHeight = blockExplorer.getBlockchainHeight();
    int maxHeight = -1;
    int peerHeight;

    System.out.println(LocalDateTime.now() + " My chain height: " + ownHeight);

    // Check chain height of peers
    for (int i = 0; i < peers.size(); i++) {

      peer = peers.get(i);
      peerHeight = peer.currentHeight();

      if (peerHeight > maxHeight) {
        maxHeight = peerHeight;
        syncToPeer = peer;
      }
    }

    System.out.println(LocalDateTime.now() + " Max chain height of peers: " + maxHeight);

    // Start sync with longest chain
    String msgType, message;
    if (maxHeight > ownHeight) {

      System.out.println(LocalDateTime.now() + " Syncing with " + syncToPeer.hostname() + " " + syncToPeer.port());

      // Reset values
      maxConsensusHeight = -1;
      numberOfBlocks = -1;

      // Initialize storedBlocks
      syncToPeer.initialiseStoredBlocks();

      // If own height is zero, skip immediately to request blocks
      if (ownHeight != 0) {
```

```
          // Start consensus process
          msgType = "1"; /* Check hash at height */
          message = ownHeight + separator + blockExplorer.getPoWByHeight(String.valueOf(ownHeight));
          syncToPeer.writer().println(msgType + msgSeparator + message);
          System.out.println(LocalDateTime.now() + " Sent: " + msgType + msgSeparator + message);

          // TODO Implement semaphore locking instead of busy waiting
          // http://stackoverflow.com/questions/8409609/java-empty-while-loop
          while (maxConsensusHeight == -1)
            System.out.println("Determining height of last consensus block...");

        } else {
          maxConsensusHeight = 0;
        }

        numberOfBlocks = maxHeight - maxConsensusHeight;

        // Request blocks
        msgType = "3";
        message = new String();
        int start = maxConsensusHeight + 1;

        for (int requestedHeight = start; requestedHeight <= maxHeight; requestedHeight++)
          message += String.valueOf(requestedHeight) + separator;

        syncToPeer.writer().println(msgType + msgSeparator + message);

        System.out.println(LocalDateTime.now() + " Sent: " + msgType + msgSeparator + message);

        // TODO Implement semaphore locking instead of busy waiting
        // http://stackoverflow.com/questions/8409609/java-empty-while-loop
        while (syncToPeer.storedBlocks().size() < numberOfBlocks)
          System.out.println("Waiting to receive requested blocks...");
```

```
    // If own height is zero, skip immediately to adding blocks
    if (ownHeight != 0) {

      // Remove blocks
      Node blockNode;
      for (int height = start; height <= ownHeight; height++) {
        blockNode = blockExplorer.getBlockNodeByHeight(String.valueOf(height));
        blockExplorer.removeBlockNode(blockNode);
        System.out.println(LocalDateTime.now() + " Removed block: " + height);
      }
    }

    // Add new blocks
    ArrayList<Block> storedBlocks = syncToPeer.storedBlocks();
    Block block;
    boolean valid = false;
    boolean isNew = false;

    for (int i = 0; i < storedBlocks.size(); i++) {

      block = storedBlocks.get(i);

      try {

        if (block.height().compareTo("1") == 0) {
          valid = true;
          isNew = true;
        } else {
          valid = block.validate(blockExplorer, utxoExplorer);
          isNew = isNewBlock(block);
        }

      } catch (BlockHeaderException e) {
        e.printStackTrace();
      } catch (MerkleTreeException e) {
```

```
          e.printStackTrace();
        } catch (TransactionException e) {
          e.printStackTrace();
        }

        if (valid & isNew) {
          blockExplorer.addNewBlock(block);
          utxoExplorer.update(block);
          System.out.println(LocalDateTime.now() + " Added block: " + (i + 1) + " of " + storedBlocks.size());
        } else {
          // TODO Break current sync process, discard stored blocks, and send a fresh request for blocks
          System.out.println(LocalDateTime.now() + " Block " + (i + 1) + " is not a valid block");
        }

      }

      // Update blockchain file
      blockExplorer.updateBlockchainFile();

      // Rebuild UTXO file
      utxoExplorer.rebuildUTXOList(blockExplorer);
    }
  }
}
```

Pong thread

```java
/**
 * Continuously broadcasts own height.
 */
class Pong extends Thread {

  public Pong() {
    super("Pong");
  }

  public void run() {

    System.out.println(LocalDateTime.now() + " Pong thread started");

    String msgType = "0";
    String height;

    while (true) {

      if (this.isInterrupted()) System.out.println(LocalDateTime.now() + " Pong is interrupted");

      try {
        height = String.valueOf(blockExplorer.getBlockchainHeight());
      } catch (NullPointerException e) {
        height = "-1";
      }

      for (int i = 0; i < toPeers.size(); i++) toPeers.get(i).println(msgType + msgSeparator + height);

      try {
        sleep(10000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Miner thread

```java
/**
 * Continuously mines new blocks.
 */
class Miner extends Thread {

  public Miner() {
    super("Miner");
  }

  public void run() {
    System.out.println(LocalDateTime.now() + " Miner thread started");
    while (true) {
      if (this.isInterrupted()) System.out.println(LocalDateTime.now() + " Miner is interrupted");
      mine();
    }
  }
}
```

Public and private methods, in alphabetical order

```java
/*
 * Broadcasts message to all peers.
 */
private void broadcast(String msgType, String message) {
  for (int i = 0; i < toPeers.size(); i++) toPeers.get(i).println(msgType + msgSeparator + message);
}
```

```java
/*
 * Returns a shallow clone of the transactions in the memory pool.
 */
private ArrayList<Transaction> cloneMempool() {
  ArrayList<Transaction> transactions = new ArrayList<Transaction>();
  for (int i = 0; i < mempool.size(); i++) transactions.add(mempool.get(i));
  return transactions;
}
```

```java
/*
 * Creates a new Peer object using the socket provided.  Adds the new
 * peer to the list of peers (for polling using BlockListener).  Adds
 * the associated PrintWriter to the list of toPeers (for broadcasting).
 */
private void connect(Socket socket) throws IOException {
  Peer peer = new Peer(socket);
  peers.add(peer);
  toPeers.add(peer.writer());
}
```

```java
/*
 * Removes any transaction which attempts to double-spend within the same transaction or
 * group of transactions.  Finalizes the transactions which remain.
 */
private ArrayList<Transaction> finalise(ArrayList<Transaction> transactions) {

  ArrayList<TransactionReference> references = new ArrayList<TransactionReference>();
  TransactionReference reference;
  boolean seen;

  Transaction tx;
  ArrayList<Input> inputs;

  // Check for double-spending in same transaction or group of transactions
  for (int i = 0; i < transactions.size(); i++) {

    tx = transactions.get(i);

    inputs = tx.inputs();
    for (int j = 0; j < inputs.size(); j++) {

      reference = inputs.get(j).reference();
      seen = seen(references, reference);

      if (seen) {
        transactions.remove(i);
```

```java
          System.out.println("Double-spending detected: transaction " + i + " has been removed");
        }
        else references.add(reference);
      }
    }

    // Validate transactions
    boolean validTransaction = false;

    for ( int k = 0; k < transactions.size(); k++) {
      tx = transactions.get(k);
      System.out.println("Validating transaction " + (k + 1) + " of " + transactions.size() + " ...");
      try {
        validTransaction = tx.validate(blockExplorer, utxoExplorer);
        if (!validTransaction) transactions.remove(tx);
      } catch (TransactionException e) {
        transactions.remove(tx);
        e.printStackTrace();
      }

    }
    return transactions;
  }
```

```java
  /*
   * Prompts the user for the hostnames and port numbers of the peers it wishes to connect to
   * TODO Re-factor so that it can handle input errors
   * TODO Figure out when it is safe to release the Scanner resources
   */
  private void findPeers() {

    String response, peerHostname;
    int peerPort;

    Scanner robot = new Scanner(System.in);
    System.out.print("Connect to peer? (Y/N): ");
    response = robot.nextLine();
```

```java

    while (response.compareTo("Y") == 0 || response.compareTo("y") == 0) {

      System.out.print("Hostname: ");
      peerHostname = robot.nextLine();

      System.out.print("Port: ");
      peerPort = Integer.valueOf(robot.nextLine());

      try {
        Socket socket = new Socket(peerHostname, peerPort);
        connect(socket);
        System.out.println(LocalDateTime.now() + " Connected to " + peerHostname + " " + peerPort);

        System.out.print("Connect to peer? (Y/N): ");
        response = robot.nextLine();

      } catch (UnknownHostException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
```

```java
  /**
   * Initializes the explorers.
   * @param blockchainFilename blockchain filename
   * @param utxoFilename UTXO filename
   */
  public void initialiseExplorers(Filename blockchainFilename, Filename utxoFilename, Filename walletFilename) {
    blockExplorer = new BlockExplorer(blockchainFilename);
    utxoExplorer = new UTXOExplorer(utxoFilename);
    walletExplorer = new WalletExplorer(walletFilename);
  }
```

```java
/*
 * Checks if the block's previous hash is the latest hash in the blockchain.
 */
private boolean isNewBlock(Block block) {
  String latestHeader = blockExplorer.getLastPoW();
  if (block.header().previousPoW().compareTo(latestHeader) == 0) return true;
  else return false;
}
```

```java
/*
 * Check if the transaction is already included in the memory pool.
 */
private boolean isNewTransaction(Transaction transaction) {

  String txAsString = transaction.toString();
  String newHash;

  SHA256 sha256 = new SHA256();
  if (txAsString != null) newHash = sha256.hashString(transaction.toString());
  else return false;

  String hash = null;
  for (int i = 0; i < mempool.size(); i++) {
    hash = sha256.hashString(mempool.get(i).toString());
    if (hash.compareTo(newHash) == 0) return false;
  }
  return true;
}
```

```
/*
 * Returns a new block after finalizing the transactions which will be included
 * (every valid transaction in the memory pool), creating a mint transaction,
 * obtaining a Merkle root of the tree of transactions (including the mint transaction),
 * and successfully finding a solution for the proof-of-work hash.
 */
private Block makeBlock(BlockExplorer blockExplorer) {

  // Clone memory pool
  ArrayList<Transaction> transactions = cloneMempool();

  // Finalize transactions
  transactions = finalise(transactions);

  // Add transaction fees to the mining reward
  int totalReward = totalFees(transactions) + REWARD;

  // Create new key pair and save to wallet
  // If block is not accepted, key pair is saved but the balance is invalid
  // (cannot be spent as there is no valid transaction reference)
  KeyPair newKeyPair = RSA512.generateKeyPair();
  RSAPublicKey publicKey = (RSAPublicKey) newKeyPair.getPublic();
  walletExplorer.save(newKeyPair, "0");

  // Create mint transaction
  Output gold = new Output(publicKey, String.valueOf(totalReward));
  gold.setOutputID(1);

  Transaction mint = new Transaction();
  mint.addOutput(gold);

  // Mint transaction is always at index 0
  transactions.add(0, mint);

  // Get Merkle root
```

```java
    String root = MerkleTree.getRoot(transactions);

    // Get previous proof-of-work
    String previousPoW = "xxxxxxxxxxxxxxx";
    while (previousPoW.compareTo("xxxxxxxxxxxxxxx") == 0) {
      try {
        previousPoW = blockExplorer.getLastPoW();
      } catch (NullPointerException npe) {
        findPeers();
        consensusBuilder.syncChain();
      }
    }

    BlockHeader header = new BlockHeader(blockExplorer.getLastPoW(), root);

    // Hash to find proof-of-work
    String pow = header.hash(difficulty);

    return new Block(header, pow, transactions);
  }
```

```java
  /**
   * Mines a block.  The block is added to the blockchain if it is a valid new
   * block and then broadcast to peers.  The UTXO list is updated and the memory
   * pool is cleared of transactions.
   */
  public void mine() {

    Block block = makeBlock(blockExplorer);

    if (isNewBlock(block)) {

      // Add block to the blockchain and broadcast to peers
      block.setHeight(blockExplorer.getBlockchainHeight() + 1);
      blockExplorer.addNewBlock(block);
```

```
            blockExplorer.updateBlockchainFile();
            broadcast("5", block.toString());

            // Update UTXO list
            utxoExplorer.update(block);
            utxoExplorer.updateUTXOFile();

            // Update wallet balance
            TransactionReference reference = new TransactionReference(block.pow(), "1", "1");
            String address = blockExplorer.outputAddress(reference);
            walletExplorer.updateBalance(address, REWARD);

        } // Block is discarded if it is not the latest block
    }
```

```
    /*
     * Re-constructs a block from the received block broadcast.  Returns a block
     * if the re-constructed block is valid; null otherwise.
     */
    private Block readBlock(String string)
      throws MalformedTransactionException, BlockHeaderException, MerkleTreeException, TransactionException {

      String[] sections = string.split("HEAD");
      String header = sections[0];
      String _header = sections[1];

      // Header
      String[] headerSections = header.split(separator);

      int height = Integer.valueOf(headerSections[0]);
      String previousPoW = headerSections[1];
      String pow = headerSections[2];
      String merkleRoot = headerSections[3];
      String difficulty = headerSections[4];
      int nonce = Integer.valueOf(headerSections[5]);
```

```
    BlockHeader blockHeader = new BlockHeader(previousPoW, merkleRoot);
    blockHeader.setDifficulty(difficulty);
    blockHeader.setNonce(nonce);

    // Meta data
    String[] numTx = _header.split("NUM");
    int numberOfTransactions = Integer.valueOf(numTx[0]);
    String _numTx = numTx[1];

    // Transactions
    String[] txs = _numTx.split("TX");
    if (txs.length != numberOfTransactions) throw new MalformedTransactionException("Mismatched number of transactions");

    ArrayList<Transaction> transactions = new ArrayList<Transaction>();
    Transaction transaction;
    for (int i = 0; i < numberOfTransactions; i++) {
      System.out.println(LocalDateTime.now() + " Reading transaction " + (i + 1) + " of " + numberOfTransactions + " ...");
      transaction = readTransaction(txs[i]);
      transactions.add(transaction);
    }

    Block reconstructedBlock = new Block(blockHeader, pow, transactions);
    reconstructedBlock.setHeight(height);
    System.out.println(LocalDateTime.now() + " Block " + reconstructedBlock.pow() +
      " received and reconstructed: "+ (reconstructedBlock.toString().compareTo(string) == 0));

    return reconstructedBlock;
  }
```

```java
/*
 * Parses a transaction message and returns a Transaction object.
 */
private Transaction readTransaction(String tx) throws MalformedTransactionException {

  Transaction transaction = new Transaction();

  int numberOfInputs, inputID;
  Input input;
  TransactionReference reference;
  String refpow, reftx, refout;

  int numberOfOutputs, outputID;
  Output output;
  String amount;

  byte[] encodedPublicKeySpecBytes;
  X509EncodedKeySpec encodedPublicKeySpec;
  RSAPublicKey publicKey = null;

  KeyFactory factory = null;
  try {
    factory = KeyFactory.getInstance("RSA");
  } catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
  }

  String inputs, outputs, signatures, in, out;
  String[] nums, parts, ins, outs, sigs, inparts, outparts;
  int txID;

  nums = tx.split("#");
  txID = Integer.valueOf(nums[0]);
  numberOfInputs = Integer.valueOf(nums[1]);
  numberOfOutputs = Integer.valueOf(nums[2]);
```

```java
    tx = nums[3];

    transaction.setTxID(txID);

    parts = tx.split("END");
    if (numberOfInputs > 0) {
      if (parts.length != 3) throw new MalformedTransactionException();
      else {
        inputs = parts[0];
        outputs = parts[1];
        signatures = parts[2];

        // Inputs
        ins = inputs.split("IN");
        if (ins.length != numberOfInputs) throw new MalformedTransactionException("Mismatched number of inputs");
        else {
          for (int j = 0; j < ins.length; j++) {

            in = ins[j];

            inparts = in.split(separator);
            inputID = Integer.valueOf(inparts[0]);
            refpow = inparts[1];
            reftx = inparts[2];
            refout = inparts[3];

            reference = new TransactionReference(refpow, reftx, refout);
            input = new Input(reference);
            input.setInputID(inputID);

            transaction.addInput(input);
          }
        }

        // Signatures
```

```java
        sigs = signatures.split("SIG");
        if (sigs.length != ins.length)
          throw new MalformedTransactionException("Number of signatures does not match number of inputs");
        else {

          transaction.initialiseSignatures(sigs.length + 1);

          for (int k = 0; k < sigs.length; k++)
            transaction.signatures()[k + 1] = BaseConverter.stringHexToDec(sigs[k]);
        }

      }
    } else /* numberOfInputs = 0 */ {
      inputs = parts[0];
      outputs = parts[1];
    }

    // Outputs
    outs = outputs.split("OUT");
    if (outs.length != numberOfOutputs) throw new MalformedTransactionException("Mismatched number of outputs");
    else {
      for (int m = 0; m < outs.length; m++) {

        out = outs[m];

        outparts = out.split(separator);
        outputID = Integer.valueOf(outparts[0]);
        encodedPublicKeySpecBytes = BaseConverter.stringHexToDec(outparts[1]);
        encodedPublicKeySpec = new X509EncodedKeySpec(encodedPublicKeySpecBytes);
        amount = outparts[2];

        try {
          publicKey = (RSAPublicKey) factory.generatePublic(encodedPublicKeySpec);
        } catch (InvalidKeySpecException e) {
          e.printStackTrace();
```

```java
      }

      output = new Output(publicKey, amount);
      output.setOutputID(outputID);

      transaction.addOutput(output);
    }
  }

  return transaction;
}
```

```java
/*
 * Returns true if this transaction reference matches any in the list of transaction
 * references.
 */
private boolean seen(ArrayList<TransactionReference> references, TransactionReference newReference) {

  TransactionReference reference;

  for (int i = 0; i < references.size(); i++) {

    reference = references.get(i);

    if (newReference.pow().compareTo(reference.pow()) == 0 &
        newReference.transactionID().compareTo(reference.transactionID()) == 0 &
        newReference.outputID().compareTo(reference.outputID()) == 0) return true;
  }
  return false;
}
```

```java
/*
 * Computes the total of fees for all finalized transactions.
 */
private int totalFees(ArrayList<Transaction> finalisedTransactions) {
  int fees = 0;
  for (int i = 0; i < finalisedTransactions.size(); i++)
    fees += finalisedTransactions.get(i).transactionFee(blockExplorer);
  return fees;
}
```

```java
/**
 * Adds a transaction to the memory pool.
 * @param transaction
 */
public void updateMempool(Transaction transaction) {
  mempool.add(transaction);
}
```

MalformedTransactionException

```java
public class MalformedTransactionException extends Exception {

  public MalformedTransactionException() {
    super();
  }

  public MalformedTransactionException(String msg) {
    super(msg);
  }

}
```

8.4.3 MerkleTree.java

Package declaration and imports

```java
package util;

import java.time.LocalDateTime;
import java.util.ArrayList;
import obj.Transaction;
```

Public methods

```java
  /**
   * Returns the hash of the root of the Merkle tree built using the transactions provided.
   * @param transactions array list of transactions
   * @return root of the Merkle tree of transactions, including the mint transaction
   */
  public static String getRoot(ArrayList<Transaction> transactions) {
    int numberOfLeaves = countLeaves(transactions.size());
    return buildTree(transactions, numberOfLeaves);
  }
```

```java
  /**
   * Checks if the Merkle tree root corresponds to the transactions
   * @param transactions
   * @param root Merkle tree root
   * @return true if the provided root matches the root from the reconstructed Merkle tree
   * @throws MerkleTreeException
   */
  public static boolean validateRoot(ArrayList<Transaction> transactions, String root) throws MerkleTreeException {
    String rebuiltRoot = getRoot(transactions);
    if (rebuiltRoot.compareTo(root) == 0) return true;
    else throw new MerkleTreeException("Root does not match transactions");
  }
```

MerkleTreeException

```java
public static class MerkleTreeException extends Exception {

  public MerkleTreeException() {
    super();
  }

  public MerkleTreeException(String msg) {
    super(msg);
  }
}
```

Private methods

```java
/*
 * Returns the minimum number of leaves that is larger than the number of transactions,
 * with the condition that the number of leaves must be a power of two.
 */
private static int countLeaves(int numberOfTransactions) {
  int powerOfTwo = 0;
  while (Math.pow(2, powerOfTwo) < numberOfTransactions) powerOfTwo++;
  return (int) Math.pow(2, powerOfTwo);
}
```

```java
/*
 * Returns the root of the Merkle tree.
 */
private static String buildTree(ArrayList<Transaction> transactions, int numberOfLeaves) {

  int numberOfTransactions = transactions.size();
  String txAsString;
  SHA256 sha256 = new SHA256();
  ArrayList<String> hashes = new ArrayList<String>();
  String hash = null;

  // Set transaction ID
  for (int i = 0; i < numberOfTransactions; i++) transactions.get(i).setTxID(i + 1);
```

```java

    // Get hash of transaction string
    String[] txsAsString = getTransactionsAsStringArray(transactions);
    for (int j = 0; j < txsAsString.length; j ++) {
      txAsString = txsAsString[j];
      hash = sha256.hashString(txAsString);
      hashes.add(hash);
    }

    // Add the last real transaction leaf to the tree until there are enough total leaves
    for (int leaf = numberOfTransactions; leaf < numberOfLeaves; leaf++) hashes.add(hash);
    while (hashes.size() > 1) hashes = fold(sha256, hashes);

    System.out.println(LocalDateTime.now() + " Merkle tree built with " + transactions.size() + " transactions");

    return hashes.get(0);
  }
```

```java
  /*
   * Ensures that the transactions are ordered according to their transaction IDs.
   */
  private static String[] getTransactionsAsStringArray(ArrayList<Transaction> transactions) {

    int numberOfTransactions = transactions.size();
    String[] txArray = new String[numberOfTransactions];

    Transaction transaction;
    int transactionID;

    for (int i = 0; i < numberOfTransactions; i++) {
      transaction = transactions.get(i);
      transactionID = Integer.valueOf(transaction.txID());
      txArray[transactionID - 1] = transaction.toString();
    }

    return txArray;
  }
```

```java
/*
 * Folds each level of the tree into half.
 */
private static ArrayList<String> fold(SHA256 sha256, ArrayList<String> hashes) {

  ArrayList<String> results = new ArrayList<String>();

  int i = 0;
  while (i < hashes.size()) {
    String concat = hashes.get(i) + hashes.get(i + 1);
    String result = sha256.hashString(concat);
    results.add(result);
    i = i + 2;
  }

  return results;
}
```