Imperial College of Science, Technology and Medicine

Department of Computing

3rd Year Individual Project Final Report

A Blockchain-based Information Management System

Author:
Jiranart Vacheesuthum
CID 00972930

Supervisor:
Dr. Konstantinos Gkoutzis

June 19, 2017

Abstract

Blockchain Technology, made famous by Bitcoin, has been a great influence on how systems are designed in recent years. Offering several strengths such as immutability, resilience and guaranteed integrity, the Blockchain architecture has been adopted by many different types of systems, even beyond digital currency platforms.

In the ever-growing world of data, storage, capacity and processing power never cease to advance. However, even with the latest security measures in place, one essence of data that still lags behind the rest is trust. With fake news from untrustworthy sources becoming more popular than facts and the truth, common sense is challenged daily and the results can sometimes be devastating. We can have the largest database and achieve the greatest efficiency with the fastest multi-processor grid, but if we cannot trust the data that reside in it, all our efforts are meaningless.

With the aim to take advantage of the benefits that the Blockchain possesses in a conventional database, we present BIMS: the Blockchain-based Information Management System. In order to prove the concept of 'fusing' these two technologies, we narrow down the data input of our prototype system. This specific system operates on semantic ontology input using RDF files, and manages to produce a distributed, immutable and integrity-guaranteed database, eventually creating an "economy based on information". We also explore how elements like proof-of-work and credits systems can be utilised to further reinforce our system.

We evaluate the system in term of its functionality compared to other similar systems, its performance, as well as the strengths and weaknesses it possesses. Finally, we summarise the achievements of this project and explore how it can be extended in the future.

Acknowledgements

My first and deepest gratitude goes to Dr.Konstantinos Gkoutzis, the supervisor of this project. It has been a pleasure and fun to work with him and, more importantly, without the time and effort he dedicated, this project wouldn't be where it is right now.

Secondly, I'd like to thank Dr.Tony Field, the second marker of this project and one of the coolest lecturers (if not THE coolest) here at Imperial College, for a fruitful discussion we had during the interim report, as well as for the time he will devote to examine this project.

Special thanks to Prof.William Knottenbelt and Iain Stewart, with whom we had a meeting and discussed various topics involving Bitcoin and Blockchain. Many elements of this report have been inspired and driven by the discussions we had with them.

Finally, to my parents and family members: thank you for your support, both mentally and financially, throughout the tough years at Imperial.

Contents

1 Introduction **6**

2 Background **8**
2.1 Bitcoin and Blockchain . . . 8
2.1.1 Blockchain Technology . . . 8
2.2 The Semantic Web . . . 9
2.2.1 Resource Description Framework . . . 9
2.2.2 SPARQL . . . 10
2.2.3 State of the Art . . . 10
2.3 Decentralised Database . . . 11
2.3.1 Peer-to-Peer Networking . . . 11
2.3.2 Distributed Hash Table . . . 12
2.4 Existing Related Work . . . 13
2.4.1 BitTorrent . . . 13
2.4.2 Solid . . . 14
2.4.3 Framework with Blockchain and Linked Data . . . 15
2.4.4 Metakosmos . . . 16
2.4.5 Steem . . . 17
2.4.6 MaidSafe . . . 17
2.4.7 Golem . . . 18
2.4.8 BigchainDB . . . 18
2.5 Tools and Libraries . . . 19
2.5.1 Neo4j . . . 19
2.5.2 RDF4J . . . 20
2.5.3 Hashcash . . . 20

3 Design **21**
3.1 The 'History' Blockchain . . . 21
3.2 The Main Blocks and Graph Generation . . . 22
3.3 The Decentralisation . . . 23
3.4 The Credits System . . . 24
3.5 User Journey . . . 26

4 Implementation **27**
4.1 History Blockchain and History Entry . . . 27
4.2 Private-Public Key Pair and Digital Signature . . . 28
4.2.1 KeyPair Generation . . . 28
4.2.2 Generating and Verifying a Digital Signature . . . 29
4.3 Blockchain Compiler and Main Block Entry . . . 31
4.3.1 BCCompiler and Compiling Algorithm . . . 31
4.3.2 MainBlockEntry . . . 33
4.3.3 Neo4j Graph Generation . . . 34
4.4 Mining and Proof of Work Algorithm . . . 37
4.4.1 Hashcash.java . . . 37
4.4.2 Hashcash Performance . . . 37

4.5 Main.java . . . 39
4.5.1 User Operations . . . 39
4.5.2 The Command Parser . . . 41
4.5.3 Parsing an Ontology with RDF4J . . . 41
4.6 Tracker Server and Networking . . . 42
4.6.1 The Tracker Server . . . 42
4.6.2 P2P Networking . . . 42

5 Testing **43**

6 Evaluation **47**
6.1 Functionality . . . 47
6.1.1 Specification Comparison . . . 47
6.1.2 Comparison with Similar Systems . . . 50
6.2 Performance . . . 51
6.2.1 Size and Scalability . . . 51
6.2.2 Speed . . . 52
6.3 Strengths and Weaknesses . . . 54

7 Conclusion and Future Work **56**
7.1 Conclusion . . . 56
7.2 Future Work . . . 56
7.2.1 Natural Language Processing . . . 56
7.2.2 State-of-the-Blockchain Hash . . . 57
7.2.3 Information Stock Exchange . . . 57
7.2.4 Constructive Proof-of-Work Algorithm . . . 57
7.2.5 User Interface and Running Partial Nodes . . . 58
7.2.6 Encryption and Private Network . . . 58
7.3 The Scalability Problem . . . 58

8 References **60**

9 Appendix **64**

1 Introduction

Bitcoin. A digital cryptocurrency that took the world by storm a mere 8 years ago. While the starting price was well under $0.01, at the time of writing a single Bitcoin has climbed the all time high record to reach over $2900[1] for the first time. The technology behind the wheel of its success is known as the Blockchain, which will be discussed later in the Background section. The Blockchain technology, in recent years, has been adapted to empower numerous applications.

This project explores how the Blockchain architecture can be beneficial to database and data management systems. First of all, the Blockchain is known to be hard to exploit, resilient, and decentralised, in contrast to a conventional database where there is a single point of failure[2]. The Blockchain provides anyone the ability to track and verify any data in systems, hence offering integrity and trust. One of the strengths that it exhibits is its inefficiency. Yes, this is a special case where inefficiency is strength. It takes a lot of processing power to produce a single block in the Blockchain, making it immutable. We will come back to the details later in the Design section on how the strengths that the Blockchain possess can be advantageous to our data management system.

Now let's assume that we have achieved the ideal result and we can have any kind of database which is based on the Blockchain: fully decentralised, secured, and integrity-guaranteed. Who ultimately uses this system? What kind of users and how will it be beneficial to them? An example use of this Blockchain would be a global collection of the hashes of all executable software files, like a SVN verification system. Users can check with this global Blockchain if the software they are about to install has been tampered with or not, if so the hash of that executable file wouldn't match the one approved by the community in the Blockchain. Or perhaps someone wishes to use it for cataloguing items with the ability to monitor, and account for, all additions and edits. There are endless usage possibilities for this conceptual system and we will discuss some more towards the conclusion of this report.

In this project, we have set out to design and build a prototype for a Blockchain-based data management system to prove that the concept is useful and feasible. We have narrowed down the experimental data we use in our prototype to semantic ontologies. The idea of a semantic ontology goes back to the late 1990s when the World Wide Web was blooming. As the WWW grew rapidly in the early 1990s, the pool of information that the Internet held far exceeded the capabilities of humans to keep record of and, shortly afterwards, the search engines were invented. The Semantic Web aimed to give common definitions to data that machines will be able to understand semantics and make searching more efficient. Nonetheless, it has not been adopted by the majority of end users for several reasons, one of which being that it requires technical knowledge and complicated steps to query a dataset.

With the employment of semantic technology at a large scale, the possibility in connecting and classifying data and their relationships across the World Wide Web emerges. For a development of such scale to happen, we require a collection of data – a massive collection, with as many contributions from as many contributors as possible. That is where this idea of Blockchain data management system comes in. Members can participate in adding useful information and ontologies into this system. As more people contribute, we have a larger dataset. If we link these data together, we would have created a large pool of connected data, an "economy based on information". In the background section we will discuss research papers on this topic.

It is important to note that the idea to integrate Blockchain characteristics with databases is not novel. There are scientific publications and products in the market that incorporate the architecture of the Blockchain with databases in different ways, some of which are discussed in the Background section of this report. This project differs by the motivation to explore the powerful impact that the Blockchain can make on data management systems, with a prototype focusing on semantic ontologies to prove that the concept works. We will examine potential issues such as efficiency and serviceability which are crucial factors of a data management system and finally evaluate ways in which it can be improved in the future.

2 Background

In this section, we briefly discuss the background knowledge and concepts required to understand this project and how each of them is related to this project. We then explore and evaluate a number of existing published work with similar objectives as well as how this project will differ from them. Finally, we introduce existing APIs and libraries that have been beneficial to the implementation of this project.

2.1 Bitcoin and Blockchain

Bitcoin, first launched in 2009 by an unknown person or group who named themselves Satoshi Nakamoto, is the most widely known open-source digital currency and payment system[3]. The unique selling point behind its success is the ability to perform peer-to-peer transactions securely without needing intervention from a third party (e.g. bank). This is enabled by the concept known as the Blockchain. This project will utilise the concept of Blockchain to store and verify data, so we will discuss it further in detail.

2.1.1 Blockchain Technology

Bitcoin's Blockchain is a distributed ledger shared by all participating nodes which keeps a record of all transactions.[4] All transactions are publicly announced and each block containing transaction records is built upon the previous block using hashes. Since each block contains a hash of the previous block, they are computationally impractical to modify, as the modifier would have to regenerate every block that precedes. This is what enables reliable peer-to-peer transactions in Bitcoin.

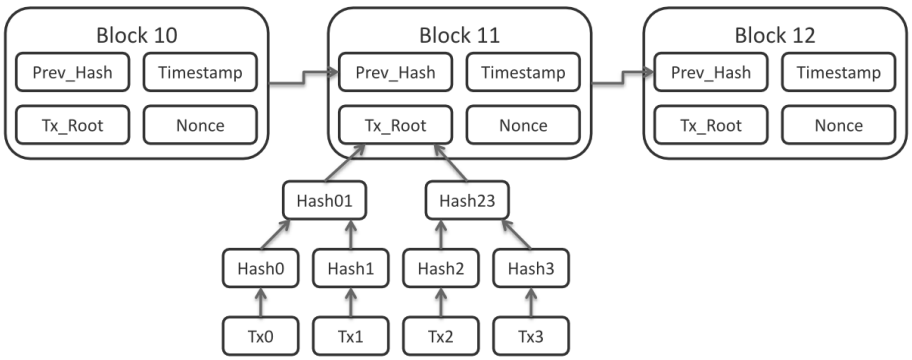


Figure 1: An illustration of Bitcoin's Blockchain [5]

Each block is constructed by *miners* who dedicate their computational power to mine blocks in order to be granted a small amount of Bitcoin as a reward. Miners use brute-force computation to verify new transactions and include them in a block, with each block being of different difficulty or *target value*. If the hash of the new block proposed is below this target value, then the mining is successful. Note that

there can only be one chain of blocks which prevail. Of course, forks can occur when different miners successfully create a block at the same time, but the system will only allow next blocks to be built on top of one block, thus abandoning the other forked block and invalidate it.

The concept of Blockchain has been widely adopted due to its simplicity, scalability, and distributed manner. Ethereum[6] is one of the most widely known Blockchain-based application platforms dealing with smart contracts. It provides a decentralised virtual machine capable of executing scripts by using a network of participating nodes. Transactions in Ethereum take place using its internal currency, known as *ether*, and are subject to fees, known as *gas*. The Ethereum platform demonstrates that Blockchain technology can be utilised outside the context of digital currency.

2.2 The Semantic Web

The Semantic Web is an extension of the Web by the World Wide Web Consortium (W3C) with a goal to standardise the format of data on the Web, allowing computers to process data according to their meaning, in order to be able to carry out more effective operations. The family of Semantic Web technologies form an environment for machines to query data and even derive a conclusion from them. We will briefly introduce two important members: the Resource Description Framework (RDF) and the SPARQL.

2.2.1 Resource Description Framework

The Resource Description Framework, or RDF[7] is a data model allowing metadata to be represented. Data are stored in *triples*, following the *subject-predicate-object* format. Simply put, it is a mechanism to describe and express resources in a way that computers can understand and process. It is the core technology for the Semantic Web to operate, providing a common format for its technologies.

```
<rdf:Description rdf:about="John">
<DC:marriedTo>Jane</DC:marriedTo>
</rdf:Description>
```

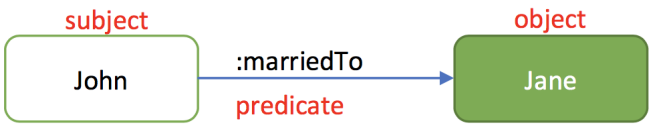


Figure 2: An example of RDF triples with a graph representation

The RDF Schema, or RDFS, provides means to define concepts like classes, datatypes, as well as define more useful relationships for machines to infer new data

from existing ones. The RDFS rule in Figure 3 describes that the :marriedTo relationship is restricted to :Human only. The RDFS also states that :Human is a subclass of :Mammal. From these rules, we can infer that John is a :Human and a :Mammal.

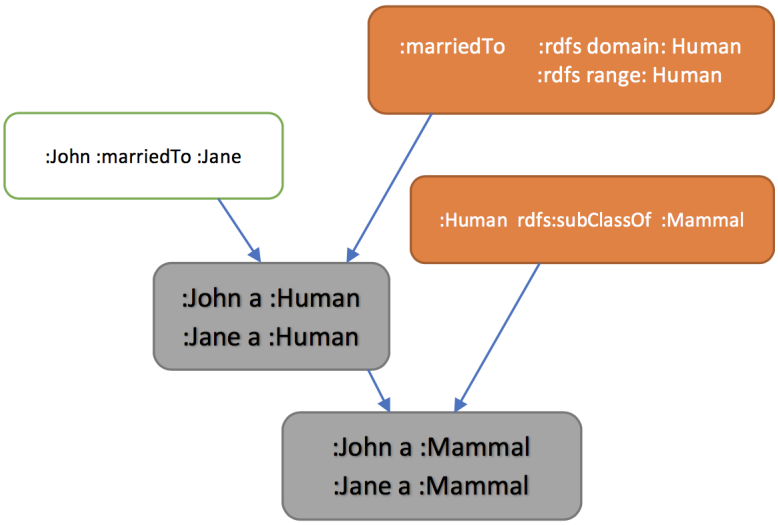


Figure 3: An example of RDF Schema and inference from existing data

2.2.2 SPARQL

SPARQL(pronounced "sparkle")[8] is a query language specifically for RDF data model. It is based on *triples* patterns like RDF, allowing the SPARQL engines to extract complex information from a pool of Semantic data and discover new relationships, producing an *inference*. From the example given in Figure 3, we can query "John is a ?species" and the engine will produce all matching answers it can arrive to by reasoning. For this instance, "John is a :Human" and "John is a :Mammal" will be returned. Both of these are not written anywhere in the data, but have been inferred through the semantics defined by RDFS.

2.2.3 State of the Art

We don't usually see or hear a lot about the Semantic Web because it is a technology often used in the background rather than openly influencing the general Web experience[9]. It's being incorporated in many fields, largely in healthcare and life sciences, as well as in a range of companies like modern search engines. Even though it possesses a lot of potential benefit for the Web, it involves the following shortfalls which prevent it from being popularised:

1. The implementations are mostly approachable to developers but not to most end users.

2. The Semantic Web embodies complicated in-depth concepts
3. Metadata currently have to be added manually, which does not allow for automation

The system this project aims to build will operate on linked data in RDF format in order to establish and depict existing relationships between data entries and organise them respectively. This will contribute to the realisation of the Semantic Web for individuals and companies that engage with the system.

2.3 Decentralised Database

A distributed or decentralised database[10] is a database in which the main processors are decoupled from its storage devices. These devices can reside at a single location or distributed across independent computers in the network. It possesses several advantages, a few of which are:

- Does not have a single point of attack or failure, thus making it less vulnerable to denial of service attacks
- High scalability; easy for expansion
- Protection of valuable data; these can be replicated and stored in many sites
- Provides continuous operation, e.g. single site failure does not affect the whole system
- Promotes ACID[11] properties of transaction

However, they also introduce a number of concerns, some of which are:

- Integrity can be computationally difficult to apply
- Security is essential, distributed fragments must be secure
- Concurrency control has to be implemented

2.3.1 Peer-to-Peer Networking

A Peer-to-Peer network[13] is an decentralised application architecture allowing tasks to load between the nodes of the network. It can be seen as a distributed database in which the common processors are the computers in the network themselves. These nodes make their resources (e.g. processing power, bandwidth) available to others in the network, instead of just receiving equally shared resources from a single site. We will discuss BitTorrent in more detail, a popularised file sharing application that uses a peer-to-peer architecture which also provides helpful concepts to this project.

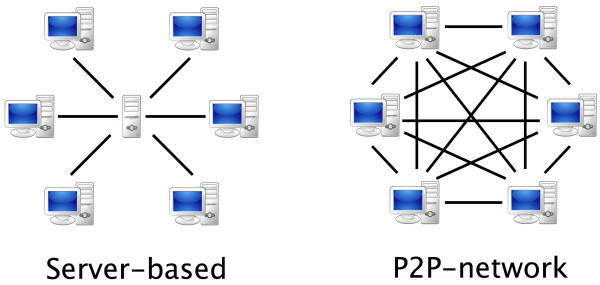


Figure 4: An illustration of resource sharing in a server-based network and a peer-to-peer network[12]

2.3.2 Distributed Hash Table

A Distributed Hash Table[14] provides a key-value pair look-up interface like a normal hash table, but the pairs are distributed across all nodes in the network. DHT utilises the concept of a decentralised database instead of storing all values in one place. Thus, it inherits all the security and performance benefits that a decentralised database embodies.

Keyspaces are split among participating nodes. Each node in the network maintains connections to several other nodes, forming an overlay network. In contrast to a traditional hash table, additions or removals of keyspaces in a node only affects the keyspaces in the neighbouring nodes, rather than having to remap the entire keyspace system. Routing in DHT is carried out by forwarding a message containing the key in search, k, to the neighbouring nodes whose ID is closest to the k. If such neighbour does not exist then we have arrived at the closest node. The algorithm is executed in a *greedy*[15] fashion allowing an average-case routing to be completed efficiently.

We will discuss a few application which take advantages of a DHT like BitTorrent as well as evaluate its usefulness to the system we are trying to build in the next section.

2.4 Existing Related Work

Below we discuss a few existing approaches in utilising blockchain and the concept of decentralisation to enhance the data managing experience, as well as potentially the Semantic Web itself. We also look into the published paper of Metakosmos, which is a possible application of the system this project aims to build.

2.4.1 BitTorrent

BitTorrent[16] is a peer-to-peer file sharing protocol. It is one of the most widely used protocols to distribute large files today and is responsible for a very high percentage of Internet traffic (25% in 2004.)[17] A group of connected computers works in a *swarm*, allowing files to be shared amongst them without the need for a central server. Large files can be broken down into small pieces and be downloaded from different nodes. A file is considered healthy if it is stored in a sufficient number of nodes. Users can connect to a swarm in order to download, known as"leeching", or to contribute bandwidth for others, known as "seeding". Seeders are rewarded with higher download speed, encouraging clients to stay connected so others can download files they hold. This is the key that makes BitTorrent stays fast and efficient.

BitTorrent clients incorporate a Distributed Hash Table technology to allow communication without a central server. Instead, they use a *tracker*, a server which keeps track of others in the swarm and shares the IP addresses of the clients as a Distributed Hash Table node, and communicates with nearby nodes, asking them to contact their nearby nodes, until the torrent location of the requested data part is found. This allows BitTorrent to become server-independent and fully decentralised.

BitTorrent has proven to be one of the most successful protocols to utilise the concept of decentralisation with file sharing systems, hence its open-source code will benefit the system we are building. This project will incorporate the concept of maintaining health of files by storing in different nodes, which is also a concept borrowed by Blockchains. Our system will also aim to establish proof-of-concept that Blockchain decentralised architecture can be beneficial to a traditional data management system.

IPFS P2P File System

The InterPlanetary File System (IPFS)[18] is a distributed file system connecting devices based on a peer-to-peer protocol, published by Juan Benet. IPFS can be seen as a repository for a single swarm in BitTorrent. It claims to provide a block storage model with high throughput and has no single point of failure due to its decentralised manner by taking advantage of Distributed Hash Tables.

The design of IPFS synthesises successful concepts from other peer-to-peer and version control protocols, including the Distributed Hash Table, BitTorrent, Git, and Self-certified Filesystems. It introduces a BitTorrent-like exchange protocol which is based on blocks, known as *BitSwap*. BitSwap embodies a credit-and-debt and ledger system to encourage nodes to share blocks to prevent only "leeching".

IPFS claims to be beneficial in many contexts as it forms a data structure which other systems can be built upon. For instance, it can be used as a linked and encrypted communication platform, a mounted global filesystem for devices in network, a personal sync folder, a versioned package manager, etc. Similar to BitTorrent, it focuses on file sharing using a decentralised approach, while this project mainly concentrates on the data management side, with an employment of semantics.

2.4.2 Solid

Solid[19] is a project supplying a set of conventions and tools for building Web applications based on Linked Data proposed by MIT and lead by the founder of the Web, Prof.Tim Berners-Lee. It utilises existing W3C standards and protocols, including the RDF data model, to empower decentralisation and privacy of data. Solid provides the following to Web applications built upon it:

- Social media capabilities such as identity management, authentication and authorisation, messaging, subscription, etc.
- A set of servers implementing Solid's specification
- A test suite for validating implementations of Solid applications
- Helper libraries and identity providers for the Solid platform

Solid aims to provide a platform that utilises linked data principles for Web application developers. It claims to improve data storage and ownership in their security aspect, by decoupling applications from the data they manage and produce. The system we will introduce takes a different approach on managing the given information, in contrast to Solid's focus on Web applications.

2.4.3 Framework with Blockchain and Linked Data

The existing published work, *Block Chain Technologies & The Semantic Web: A Framework for Symbiotic Development*[20], demonstrates how Bitcoin's Blockchain technology and the Semantic Web can contribute to the development of each other in different contexts. The same paper also introduced potential applications in combining the two technologies. For example, it can reinvent the accreditation and certification systems with smart contracts, as well as provide a universal distributed ledger for industry transactions.

Since the aforementioned work explores both the Semantic Web and the Blockchain side, it contains a number of observations which are directly useful for our project. On the Semantic Web side, it's been addressed that URIs used to identify resources possess weaknesses including:

- Centralisation: generating valid identifiers rely on a centralised Domain Name Server (DNS) system
- Persistence: events can cause the validity of identifiers to cease

Hence, we seek an ideal identifier system that can fulfil the following requirements:

- Security: authenticity of content should be ensure
- Human-readability: there are systems which give an identifier with no meaning for humans, making it hard to be remembered; this should not be the case
- Decentralisation: the system should not possess a single, centralised, point of failure

These can be partially implemented with utilisation of Blockchain technology. One example of an existing approach to this concept is *Namecoin*, a fork of Bitcoin which enables users to bypass DNS providers.

Moreover, the blockchain allows data to be stored within it. This is one of the elements this project will concentrate on and take advantage of. We will look into two methods mentioned in this paper of encoding data into Blockchain, including:

1. Multisig Address: addresses where more than a single public/private key signature is required to redeem the value

2. OP_RETURN: a command in Bitcoin's Blockchain which allows addition of metadata of up to 40 bytes to be stored on the block

This work is an attempt to combine Blockchain technology with the Semantic Web for mutual benefit. Our aim is to demonstrate how these technologies can be employed by everyday life systems, rather than just building a core management system for application platforms to be built upon.

2.4.4 Metakosmos

Moving from Folksonomies to Taxonomies: Using the Social Web and 3D to Build an Unlimited Semantic Ontology[21] is a published work by Dr.Konstantinos Gkoutzis, who is the supervisor of this project. The paper proposed a Social Web system working on top of an unlimited Semantic Ontology with a 3D Web interface called *Metakosmos*. Participators will form a community which drives the system by adding new linked data as well as editing existing ones, creating community-defined taxonomies. A reputation system allows users to up-vote and down-vote how well each published piece of information is categorised in order to maintain data quality.

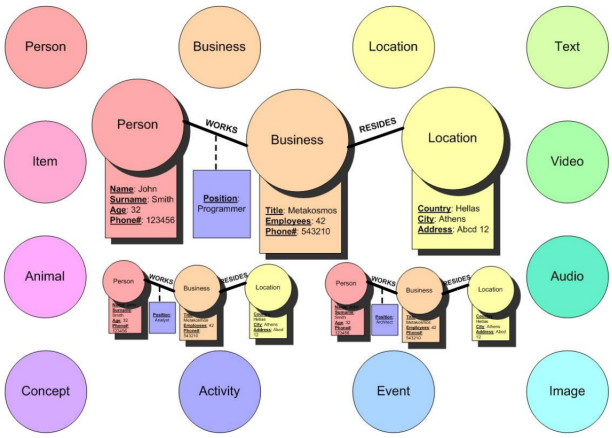


Figure 5: A searching example with Semantic Synthesis; results presented as linked objects[22]

The system uses a method called "Semantic Synthesis" allowing users to search for, on top of keyword, the association between objects they consider meaningful (in addition to keyword searching). For example in Figure 5, one can look for a *person* working in a *business* at a *location* and specify extra details for each; e.g. only search for location "United Kingdom, London, SW7." The system can produce results which are of more relevance or of higher popularity. In order for this mechanism to operate, data must be treated as objects and contained in a linked manner within a unified system, which is what this project aims to create.

2.4.5 Steem

Steem[23] is a blockchain-based social media platform with cryptocurrency rewards. It allows users to contribute to the community by creating and curating contents on the site, and to get rewarded for doing so in specific manners; e.g. when posted content is valued by others. The currency, like other cryptocurrencies, can be traded, mined, and held.

Steem rewards come in two froms, the Steem dollars and the Steem power. The Steem power gives the owner the ability to curate more content and earn additional Steem rewards, while the Steem dollars is a cryptocurrency holding monetary value which can then be spent, traded, or remitted.

The aim of creating a 'community' for people who value contents makes Steem the most similar existing project yet to the system we are about to introduce in this report. At the time of writing it has a market cap in cryptocurrency of over $546 million[24]. Although our system does not yet reward users for creating good content, it aims to do so in the future. The system we are trying to build, in this specific case, focuses on adding ontologies and mapping relationships between entities, rather than posting threads on social media like Steem.

2.4.6 MaidSafe

MaidSafe[25] claims that the cloud storage we use today is unencrypted and easily accessible to the service provider. This is a big target for hackers to steal our personal data and exploit them. MaidSafe creates a distributed network formed by unused computing power and storage with no middle-man. It allows users to upload their files, fragment and encrypt them before storing them in different nodes in the network. Only the owner has the credentials to make them available as one again. Users get rewarded for their storage contributions in the form of Safe coins, which

has a market value that can be traded just like Bitcoins.

Despite the fact that it is currently operating on an Alpha release, MaidSafe currently has a market cap of over $260 million[24]. MaidSafe and this project share the insight to remove the single point of failure of database systems. The major difference to the system we are aiming to create is that the participants in our network contribute useful data for the entire community to share, rather than storing private encrypted files.

2.4.7 Golem

Golem[26] intends to create an economy of computing power, where users can "rent" other users' machine power to perform computations, ranging from rendering computer graphics, to Big Data analysis, within matters of seconds and minutes instead of days. It can be seen as a decentralised supercomputer formed by combining spare computing power from every user's machine(s). Users can thus earn money by sharing their computing power. Again, its architecture is based on a decentralised and scalable peer-to-peer protocol.

Currently, Golem possesses a Crytocurrency market capitalisation of $462 million[24], making it one of the big names in the market right now. In contrast to Golem, our main objective is to integrate and share information rather than share computational power.

2.4.8 BigchainDB

BIGCHAINDB

BigchainDB[27] is a decentralised database which has been enhanced with enhanced with the beneficial characteristics of a Blockchain, promising linear scaling in throughput and capacity with the number of nodes, as well as efficient querying. It boasts a performance of over one million writes per second and very flexible and customizable assets. Its architecture is complimentary to decentralised file systems like the InterPlanetary File System (IPFS). Similar to other distributed systems, it claims to possess immutability, no central authority, high throughput and capacity, and low latency.

While there are several similarities to our project, the BigchainDB is designed for use cases in industry, organisations and developers who seek a scalable and immutable database which operates in a decentralised manner. Our project aims to create a community-driven database which provide users mutual benefits rather than for merely organisation or managerial uses.

2.5 Tools and Libraries

Here we briefly introduce existing open libraries and tools that have been useful in implementing the project's specifications.

2.5.1 Neo4j

Neo4j[28] is one of the first popularised graph database services. It offers a highly-scalable native database with a processing engine for users to query data and relationships efficiently and in real time. Neo4j's engine comes with an interactive interface and its own powerful query language that allows for efficient querying and updating of graphs known as *Cypher*.

In relation to this project, Neo4j will be used to visualise the entries and relationships between each of them within the entire system. We will be able to see which node is popular and even filter only the nodes attributes that we are interested in; e.g. we might only want to see entries with higher popularity and more investments. It also provides us a convenient Java API to build and search from the graph within a Java application. More details will follow on this topic in later sections.

An alternative to Neo4j is the Ontotext's GraphDB[29], which is a semantic graph database used for managing, storing, and querying structured data. Even though it is a well-known graph database that meets W3C defined standards adopted by the likes of BBC, we discovered that Neo4j's API is more appropriate for the uses of this project which aims to create and iterate through nodes and relationships, as well as the ability to be able to query real-time data. Hence we focused on Neo4j for the creation of this prototype.

2.5.2 RDF4J

Eclipse RDF4J[30] is a Java framework that handles RDF data. RDF4J can create, parse, infer, and query with Linked Data and RDF triple stores.

In this prototype we use RDF4J to parse the input from an ontology XML file (.RDF format).It provides us the methods to extract models out of an ontology, differentiate between subject, predicate and object of triple stores, and add each of these, one by one, into our management system.

2.5.3 Hashcash

Hashcash[31] is a proof-of-work algorithm used to prevent denial of service attacks in systems. It is also well-known for being the mining algorithm used in Bitcoin[32]. The idea behind Hashcash is the security property of crytographic hashes being hard to invert. It takes the CPU some time to produce a valid Hashcash stamp with respect to its level – the higher being the longer.

One very popular use of Hashcash is to prevent spam in email, since spammers will have to produce a stamp for each and every recipient which should take a long time, but does not prevent normal users from performing their daily-life emailing.

We will be using Hashcash as a mining algorithm, where users will have to devote CPU time and memory into computing a valid stamp in order to gain tokens that can be used in our system. This is expected to prevent (or at least discourage) spammers from flooding the system with dummy entries aiming to cause a denial of service attack.

3 Design

In this section we will discuss how the prototype is designed and its operations, as well as justify the choices we made – from its conception to the implementation of the prototype. We will examine how each component in the system interacts with each other in the big picture, with an aid of an architectural diagram. From this point onwards, we will address the system as **BIMS**, short for Blockchain-based Information Management System.

3.1 The 'History' Blockchain

As mentioned in the previous section, the Blockchain is reputable for its immutable property. Since it is computationally infeasible to revert it, storing data directly in the blocks of the blockchain probably does not fit the requirements of most systems. What should happen if a user wishes to remove an entry? We definitely do not want anyone being given permission to manipulate the established blocks in any way since that would compromise the entire blockchain. Thus, our decision to make the blockchain append-only sounds reasonable.

By making our Blockchain store the *history of all the transactions*, instead of just the data, we allow it to become append-only and also achieve provable transparency in the manipulation of the data. Any activities of adding, removing, mining, etc. will be tracked and recorded with user IDs uniquely identified by specific keys. Each user receives their own *public and private key pair*, used for each of their transactions. A *digital signature* is produced for each user using their private key and details of the instruction. The signature is recorded within the block and it is easily verifiable by other users that the action belongs to the user who owns the corresponding public key. Only the person who possesses the private key has the capability of producing a valid digital signature, thus making every transaction immune to spoofing. In this way, a user cannot refuse their activities, since they are all visible and openly provable to anyone who can access the blockchain.

Now that we have established the benefits of storing the history of the transactions in the Blockchain, it is time to consider the database itself. Having to distribute both the database and the Blockchain containing the history means more work, lots of network traffic, the need for more capacity, and undoubtedly lower performance. Fortunately, we have eliminated the need of having to distribute the actual database since it can be derived from its history! If we run through the history records starting from time zero, i.e. from the very first block to the latest, we can reconstruct the current state of the database to a local Database Management System (DBMS) of our choice. Every node should be able to *compile* the history and create a local copy of the database. As more and more transactions are added and distributed, users can retrieve and execute those additional transactions locally to update their copy as well. For the rest of this report, we will refer to the history of transactions recorded in the Blockchain as the **History Blockchain**.

3.2 The Main Blocks and Graph Generation

The compiled result of the History Blockchain is a series of storage objects defined by the system, which we will refer to as the **main blocks**. Note that the structure of these main blocks differs from the Blockchain and thus they do not provide any of the properties of the Blockchain. After the compilation, the BIMS software can parse these retrieved main blocks to generate a graph database, assisting the users by offering a visualisation of the current state. An architectural diagram of the interactions in the system is shown in Figure 6. The graph database service is currently provided by Neo4j in the prototype. With its help, users can easily query the graph to find something specific they might be looking for. More technical details will be covered later in the implementation section.

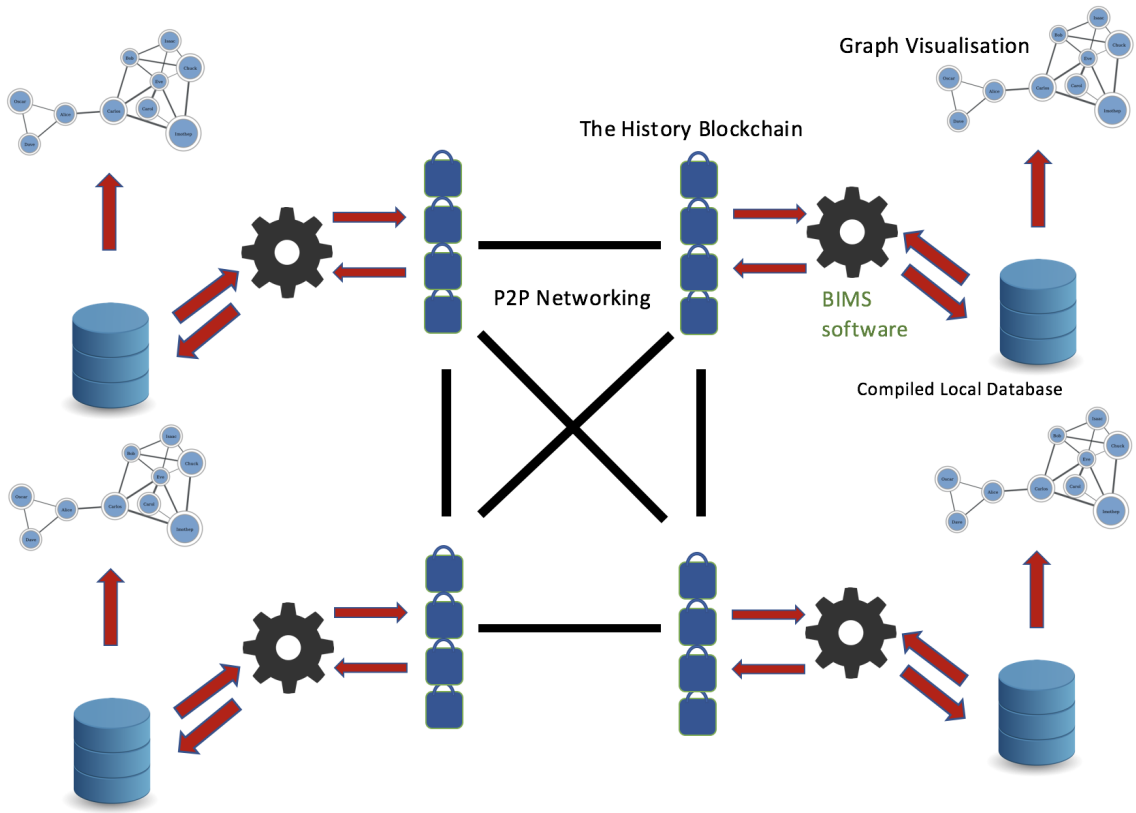


Figure 6: An illustration of how each component in the system interacts with each other

3.3 The Decentralisation

The History Blockchain is required to be distributed and the system should operate without a server or a single point of failure. We want each participant in the system to communicate directly to each other. The current participants can broadcast an update or request for the latest changes to the History Blockchain, while a new participant who just joined the network should be able to get connected to the network of 'peers' and request for an entire copy of the latest History Blockchain.

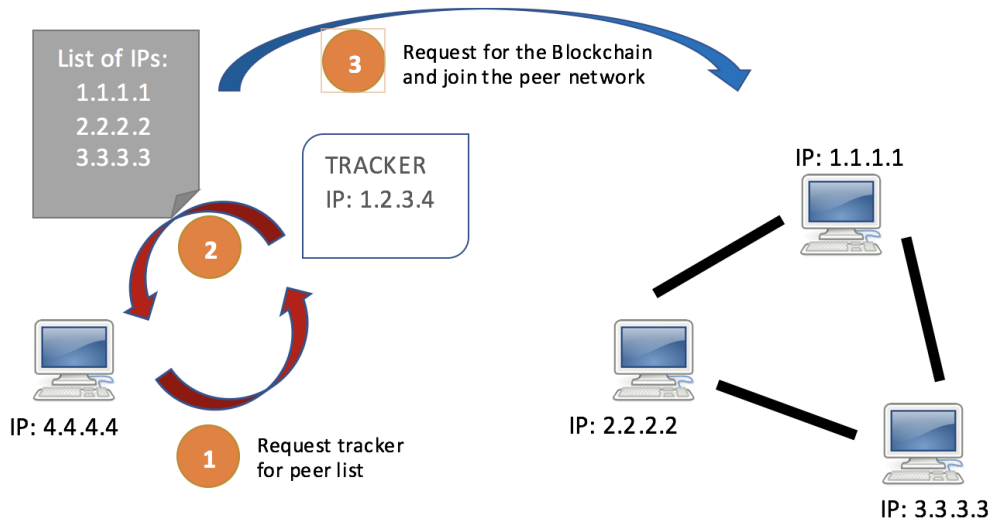


Figure 7: An illustration of how a new participant requests to join the network using the tracker

BIMS have adopted a similar peer-to-peer system to the BitTorrent protocol's 'tracker' system[33]. We have a *tracker* server running, with its address known to the public, ready to accept connections to new participants. The tracker maintains a record of the network's participants' IP addresses and port numbers. Upon request, the file containing these information is sent to the joining node, and the details of this node are added to the tracker's list. The joining participant goes through each IP address in the list that was obtained and starts sending requests for a copy of the History Blockchain until it succeeds in retrieving at least one. The nodes that have been contacted can then log the address of the new node in their list of peers. Now, the new participant has successfully joined the P2P network. Although this approach seems like the tracker host is a single point of failure, the peers who are already part of this tracker list can still operate without the need of the main tracker, without any issue. Thus, one way to mitigate this weakness is by maintaining a regularly-updated list of participants' addresses which is publicly available online under many mirror domains, so that the main tracker's website do not represent a single point of failure. This file can then be loaded directly and the new node can get into the network without the obligation to contact the main tracker directly. Potentially, the nodes can become trackers themselves, thus offering the peer list

they already maintain.

Every node in the system operates as a server and a client. As long as the software is running, any participants can establish a connection to ask for a copy of the History Blockchain or provide updates. When the user compiles the History Blockchain, the new set of transactions he/she added will be broadcasted to the list of peers the node maintains. Every node will apply this 'update' at the beginning of every program execution. This can result in a slightly different ordering of the transactions in copies of the History Blockchain but it does not matter! The transactions can be of any order without affecting the compiled result since they are the 'history'. For example, if user A and B happens to add the same entry at the same time, any of those can come first in the Blockchain, after compilation, the two will be assembled together into 1 main block with 2 investers, A and B, etc.

3.4 The Credits System

During compilation, each transaction is processed in order, one after another. There are no limits on what users can add into the History Blockchain since the system does not aim to actively and dynamically maintain the current state of the compiled database and thus does not know, nor check, the internal structure of the compiled results. Transactions that do not make sense during the compilation of the History Blockchain, like, for example, an attempt to interact with an entry that does not exist at that time, are ignored completely. However, this still presents an opportunity for spammers to flood the system with meaningless transactions which, even though will be ignored during the compilation phase, they can still lead to denial of service attacks and many more issues, like uncontrollable size increase of the blockchain. To prevent this from happening, the system runs its own credits system – similar to a digital currency.

The BIMS credits system requires users to spend their credits, known as **Koins**, for each and every transaction. Koins can be mined from running a Proof-of-Work algorithm which is integrated into the system and, not only demands CPU time and memory to process but also, enables anyone to be able to verify that the work has been calculated correctly. Each mining process, once completed, is recorded into the History Blockchain, making it available for other users to verify. In data manipulation transactions, users specify the number of Koins they will be spending. This implies that, at any state, anyone can calculate how many Koins each user has accumulated and/or spent, maintaining transparency in the BIMS credits system.

This credits system also gives data in BIMS a factor of 'importance'. Each entry in the database can be of different weight – measured in the number of credits 'invested' in it, hence when adding a new entry, each user is urged to *invest* as many Koins as he or she feels is appropriate (or can afford). Since each Koin takes time and power to be mined, it holds a token of *value*. Other users who concur on the importance of that entry can sign the entry with any amount of Koins they desire.

All these addition and re-signing instructions are 'charged' with a small fee which is kept by our system. The reverse also applies: users are permitted to remove (i.e. withdraw) the investments they have made to an entry at any time, in order to recover the Koins they have spent to spent elsewhere, again with a small fee which is kept by our system. To summarise, every transaction in the system requires a small amount of credits to be given as a fee to our system, in order to prevent spammers from flooding the blockchain – as mentioned above, but also in order for our system to maintain control and a healthy input stream. Since the number of Koins 'invested' in an entry is a property, it can filter and narrow the focus of this massive database to only the important (i.e. valuable) parts: the counterproductive or unpopular entries will receive no interests and can be filtered away when visualising the database through graph database service.

3.5 User Journey

We will briefly go through how this "economy based on information" using the BIMS prototype system operates. Assuming that there already are several participants in the network:

1. A new participant downloads the BIMS software and connects to the network through the tracker; their new private public keypair is automatically generated upon initial run

2. The participant receives a copy of the History Blockchain in its current state and compiles a local copy of the database in his/her machine

3. He/she can visualise the state of the graph with Neo4j graph database service to see the data/ontologies that have been added and their relationships

4. In order to manipulate the database, the participant must run the Proof-of-Work algorithm to accumulate an amount of Koins first if they want their additions to be processed immediately

5. There are three *commands* the participant can use to manipulate the database by adding transactions to the History Blockchain, including *ADD*, *SIGN*, and *REMOVE*. He/she specifies the data to be added/signed/removed together with the number of Koins to use. A digital signature is produced upon a successful record and the transaction is appended to his/her **local copy** of the History Blockchain.

6. The system can also take an ontology input in .RDF file format, which adds all entities and relationships contained within the ontology one by one and invest 1 Koin in each. Note that the user must have mined a sufficient number of Koins to be able to do so.

7. The participant is free to compile again to see results in graph or add more transactions however he/she likes as long as all the requirements are met

8. The appended transactions are synchronised and broadcasted to all other participants to update the state of the History Blockchain

4 Implementation

Now that we have an idea of how the each component of the system interacts to create an 'economy', we will delve into how BIMS operates at a deeper level. We will cover the technical details, the algorithms used in the prototype, and how the tools and libraries we mentioned in Section 2.5 participate in the system.

4.1 History Blockchain and History Entry

The *HistoryBC* (History Blockchain) class provides a constructor for a HistoryBC object which maintains a list of *HistoryEntry* objects. Both classes implement the *Serializable* interface, so that it can be serialised into a file and stored locally in a disk. Upon loading the file when the software is loaded, we store the state of the History Blockchain in the user's local machine. The *getInstance*() method uses a static counter in *HistoryBC* to restrict the number of instances of a *HistoryBC* object to 1.

A *HistoryEntry* object contains the following fields:

- A *Command*, an Enum specifying the transaction to be applied to the History Blockchain. The valid values are ADD, SIGN, REMV, MINED, and LINK. ADD inputs the specified data into the database with the specified number of Koins. For the prototype, the minimum amount that can be spent on every transaction is 1 Koin. SIGN allows a user to invest in other users' entries with the specified amount of Koins. If 2 users add the same data, either by chance or intentionally, the second add will have the same effect as a SIGN transaction, handled by the compiler. REMV removes an investment from an entry, if any; it also requires 1 Koin fee to operate to prevent malicious users 'flooding' the Blockchain with REMV transactions. LINK is used by the system to depict relationships between data entries, like for example in the graph from the RDF triple stores. More details will be covered later in this section.

- *Data*, which can either be in form of String (atomic data) or an array of Strings. In the case of proof-of-work, this is the Hashcash stamp value. The class Data also implements *Serializable* for state maintaining purposes.

- An *int* 'ref', which records a reference number for a block related to this transaction - if any. A transaction asking to SIGN or REMV can be provided a referential index instead of the actual data itself.

- An *int* 'koins', the number of Koins involved in the transaction

- A *byte[]* 'signature', a verifiable digital signature produced by the user's private key as a proof of validity for this transaction. This is produced upon initialising by the function *generateSig()* called in every constructor of *HistoryEntry*. We will discuss the details of producing and verifying a digital signature later in this section.

- The *PublicKey* of the user who initiated the transaction
- A *Data* secondData, which is used by the system for linking data entries (e.g. for RDF triple store purposes)

4.2 Private-Public Key Pair and Digital Signature

It is a requirement for each user to have a unique public and private key pair in order to participate in the system. As stated in Section 3.1, the public key is recorded in each transaction to track user activity, and a digital signature is produced using the private key to fend off any malicious users trying to spoof the system. Therefore, the signature generated must be accessible to anyone who wishes to verify it. These functionalities are provided by the JDK Security API(*java.security*[34]).

4.2.1 KeyPair Generation

A *KeyPairGenerator* object implements a service to create a new *KeyPair* with respect to the specified algorithm, provider, and degree of randomness. Since we are using the key pairs to produce digital signatures, we use the *DSA* or Digital Signature Algorithm and the JDK SUN provider, which is the built-in default provider for the DSA algorithm. We also employ the *SecureRandom*, which is a cryptographically strong random number generator included in the security API, to add a degree of randomness to the generation, reducing the chance of duplicated *KeyPairs* to almost zero. This is specified in the *initialize*() function together with the *keysize* in bits. We decided to use the minimum key size, 512 bits, which should be more than sufficient to prevent collisions in the prototype system and can also help with reducing the storage cost when recording the public key in each transaction.

```
//private public key pair generation --------------------------------

KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");

keyGen.initialize(512, random);

KeyPair pair = keyGen.generateKeyPair();
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();
```

4.2.2 Generating and Verifying a Digital Signature

The keys generated are passed into the functions that appends a transaction to the History Blockchain. When each HistoryEntry is constructed, *generateSig(PrivateKey priv)* in *HistoryEntry* class is invoked to produce a digital signature using the details of that transaction, along with the private key that is passed into the constructor. A *Signature* object requires the private key and the bytes(data) to be signed to produce a verifiable signature (of type byte[]) upon calling *sign*().

```
private byte[] generateSig(PrivateKey priv) {

    Signature dsa;

    try {
        dsa = Signature.getInstance("SHA1withDSA", "SUN");

        //initialise with user's private key
        dsa.initSign(priv);
        String s;

        if (this.c == Command.MINED) {
            //case for proof of work
            s = this.c + this.d.toString();
        } else if (d == null) {
            //case for signing and removing where users specify
            //index of data instead of the data itself
            s = this.c.toString() + this.ref + this.koins;
        } else {
            //regular data entry
            s = this.c + this.d.toString() +this.ref + this.koins;
        }

         //convert into bytes to be signed
        byte[] toSign = s.getBytes();

        //produce a digital signature with respect to the details of the transaction
        dsa.update(toSign);
        return dsa.sign();

    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Once a digital signature is produced, it is recorded in a History Entry. If a user wishes to verify its legitimacy, he or she only needs the public key of the user and the data the signature is signed with, both of which are publicly available in the system. A *Signature* object can be initialised for verification purposes. The $verify(Signatures)$ method returns a boolean indicating whether the signature matches the data and public key provided. The method *verifySig(byte[] signature, byte[] data, PublicKey pubKey)* implements the procedure but is not currently available for users to invoke through the user interface. Participants who wish to verify a signature can simply run the same algorithm locally after compiling their local database.

```
// method to verify signature
/*
 * params:  signature - signature to be verified
 *          data - details of transaction which produced this signature
 *          pubKey - public key of the user who produced this signature
 */
public boolean verifySig(byte[] signature, byte[] data, PublicKey pubKey) {
    try {
        Signature sig = Signature.getInstance("SHA1withDSA", "SUN");
        sig.initVerify(pubKey);
        sig.update(data);
        return sig.verify(signature);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}
```

4.3 Blockchain Compiler and Main Block Entry

4.3.1 BCCompiler and Compiling Algorithm

The *BCCompiler* class implements the function *compile(HistoryBC history)* that produces the local data store or the main blocks. Upon a *BCCompiler* object constructed, an empty ArrayList of *MainBlockEntry* to be populated when *compile()* is invoked with a valid History Blockchain. The *BCCompiler* also keeps track of a 'last_updated' *Date* which indicates the time when the current version was compiled and *int* last_index which is the index of the last HistoryEntry compiled.

The function *compile(HistoryBC history)* is described in pseudo code in Algorithm 1. It takes the list of $HistoryEntry$ from the History Blockchain, iterates through the list, continuing from the last_index and invokes a sub helper function according to the $Command$ in each entry. These helper functions carry out necessary procedures before creating and/or manipulating the main blocks in each specific case.

- $compileAdd()$ first searches for any duplication with the data the entry is attempting to add. If there is no match, the data are added to main blocks; if there is a duplication, it will proceed and invest the specified amount of Koins into the existing entry instead.
- $compileSignRemv()$ is used for both SIGN and REMV commands since they follow a similar procedure. It retrieves the referential index of the data the transaction is involved with, checks if it is an existing and valid index, before investing (SIGN) or removing investment (REMV) from that particular entry for that user in the main blocks.
- $addMine()$ simply adds a proof-of-work entry into the main block
- $addLink()$ adds a special main block which specifies 2 main blocks to be linked by a relationship. As mentioned above, this is currently only used by the system when a user adds a previously interlinked RDF ontology into the History Blockchain.

Algorithm 1 Compile a History Blockchain

```
List<MainBlockEntry>entries
Date last_update
int last_index
procedure COMPILE(HistoryBC)
    H = HistoryBC.getList() from last_index to end of list
    for each HistoryEntry e in H do
        command = e.getCommand()
        read(command)
        if ADD then
            compileAdd(e)
        end if
        if REMVorSIGN then
            compileSignRemv(e)
        end if
        if MINED then
            addMine(e)
        end if
        if LINK then
            addLink(e)
        end if
    end for
    update(last_update)
    update(last_index)
end procedure
```

4.3.2 MainBlockEntry

A series of *MainBlockEntry* objects maintain with the *BCCompiler* make up the current state of the main blocks. A *MainBlockEntry* object encompasses the following fields:

- *Data* 'data', or the data that this block represents.
- *List<Inv>* 'investments'. A list of *Inv* objects, keeping track of the investments each user has made to this entry. Each *Inv* object contains
  - *byte*[] 'pubkey', the encoded public key of the investor using
  - *byte*[] 'sig' the digital signature that can be used to verified the legitimacy of this investment
  - *int* 'koins', the amount of Koins this user has invested in this main block
- *int* 'totalKoins'. The sum value of all the Koins invested in this entry by all the users.
- *int* 'index', or the index of this main block
- *String* 'pubkey'. This is the public key which is used for the proof-of-work entry;. It is encoded using $Base64.getEncoder().encodeToString()$.
- *Data* 'secondData'. A Data object used solely for linking RDF entities together. An entry with a not-null secondData indicates that when a graph is created, a relationship should be initiated between the 'Data' and 'secondData'.
- static *int* 'counter'. A static counter used to allocate the index number of each main block.

MainBlockEntry implements the function *invest*() and *removeInvest*() to adjust the investments in each block. *removeInvest(byte[] pub, int koins)* searches through *investments* for a matching public key before removing the specified amount of Koins from it. If the user has invested none or insufficient Koins in the block, the system ignores the transaction.

```java
public void removeInvest(byte[] pub, int koins) {
    int index = searchPub(pub);
    if (index > -1) {
        int k = investments.get(index).getKoins();
        if (k == koins) {                //removing all koins
            totalKoins -= k;
            investments.remove(index);
        } else if (k > koins) {         //removing only some investments
            totalKoins -= koins;
            investments.get(index).setKoins(k - koins);
        } else {                        //command asks to remove more than invested, ignore
            System.out.println("User has invested only " + k + " koins in this block");
        }
    } else {
        System.out.println("User has not invest any koins in this block, unable to remove");
    }

}
```

4.3.3 Neo4j Graph Generation

Once we have a populated list of *MainBlockEntry* objects, we can use it to generate a graph database for a better visualisation of the database state. The Neo4j Native Java API[35] is capable of creating labelled graph nodes, informative node properties, as well as relationships.

GraphDatabaseService provides useful operations related to Neo4j nodes. In this instance we use it to create new nodes with *createNode(Label label)* where the *Label* object is used to identify the 'group' this node belongs to. One node can belong to many different groups, but since we do not require any grouping at the moment, we will use Label "ALL" for every node in the system for now , which should simplify the operations in the Neo4j Web interface. We will look into this benefit later after we have created a graph.

Now that we can create a *Node* for Neo4j, we can set properties for each node by using *Node.setProperty(String key, Object value)*. Different types of nodes in our system will have different properties. A normal data-store node holds the following properties:

- *index*, the index of the block this node represents
- *name*, the String version of *Data* which the block stores
- *total Koins invested* in the block
- *investors*, the list of users who have invested in this block along with the amount of Koins they invested

A proof-of-work block will create a mining node with the following properties:

- *index* of the block
- *PublicKey* of the miner who completed the proof-of-work

- $STAMP$, the Hashcash stamp produced upon completing the mining, which can be checked by anyone who wishes to verify it. We will discuss this in section 4.4.

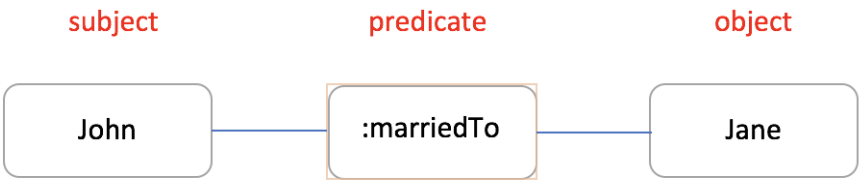
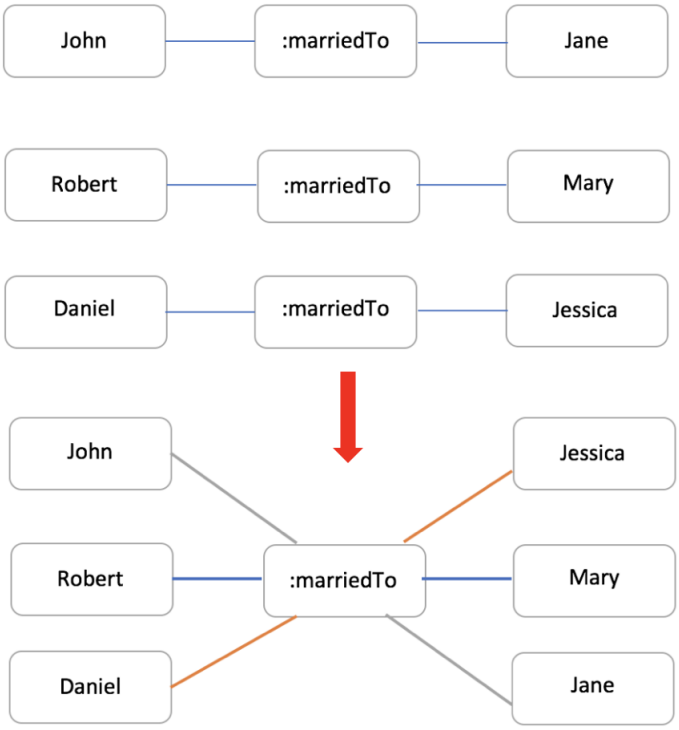


Figure 8: An example of triple store representation in the generated Neo4j graph in contrast to Figure 3 in Section 2.2.1

A main block holding a non-null *secondData* field is used for linking entities and will not create a new node in the graph. Instead, the system finds the matching nodes for the two Data entries and create a relationship between them. *GraphDatabaseService*'s method *findNode(Label label, String key, Object value)* allows the system to search for a graph N ode within the group 'label' with a property 'key' matching the value 'value'. We use this method to search for the nodes with the matching names and then use *createRelationshipTo(Node otherNode, RelationshipType type)* to link them in the graph we produced. The *relationship* can but does not hold any property or label, as the 'predicate' in the subject-predicate-object triple store is also treated as a Data entry in BIMS, as demonstrated in Figure 8.

This makes the relationships 'reusable'. Since we are collecting multiple ontologies in the system, a predicate is very likely to be used more than once and we only need one data node for each, instead of creating many identical labelled relationships. Not only we have managed to reduce the size of the main block, but we also simplified the graph representation. Instead of having a new labelled relationship for every predicate, we can observe which predicate is used more often. For example:



The function *genGraph*() in *BCCompiler* ompiler implements all the procedures to produce a graph directory which can be read and executed by the Neo4j software. The implementation can be found in the Appendix B Figure 15. We will look at an example of a Neo4j graph produced in Section 5.

4.4 Mining and Proof of Work Algorithm

4.4.1 Hashcash.java

The implementation of the Hashcash library in Java is the courtesy of Greg Rubin[36]. The static function *mintCash()* carries out the proof-of-work algorithm. Once invoked, it will produce a *HashCash* object, which embodies a Hashcash 'stamp'. There are several variations of *mintCash()*; we adopted the simplest version which takes 2 arguments: a String *resource* and an int *value*, where resource is the string to be encoded in the Hashcash and the value can be treated as the 'difficulty' of the hash.

```
//function that executes and mint hashcash in order to accumulate koins for users
    //returns the new current no of koins
    //pass historybc in to add the proof of mining

    //performance test for hashcash
    public static void mineKoins(String string) throws NoSuchAlgorithmException {
        int avg = 0;
        int level =23;
        for(int i =0; i < 10; i++) {
            Stopwatch st = Stopwatch.createStarted();
            HashCash h = HashCash.mintCash(string, level);
            st.stop();
            System.out.println("millisec: " + st.elapsed(TimeUnit.MILLISECONDS));
            avg += st.elapsed(TimeUnit.MILLISECONDS);
        }
        System.out.println("avg = " + avg/10);
        return;
    }
```

Figure 9: Performance testing code for producing Hashcash at different levels

4.4.2 Hashcash Performance

Value must be between 0 and hash length, which is specified to be 160 in this library. The algorithm requires the "number of leading zeros" in the hash produced to be more than the value specified, therefore the higher the value, the longer it takes to produce a valid Hashcash for any resource string. While searching for the 'ideal' level, we ran a test function with a stopwatch to observe the average time it took to compute a Hashcash for values 1 to 23 with test resource string "hello". Each value is run 10 times to observe the mean time, plotted in Figure 10.

We observed that the time required to compute a valid Hashcash grows exponentially with level. The test was carried out on a MacBook Pro on macOS Sierra version 10.12.2 with 2.3GHz dual-core Intel Core i5, Turbo Boost up to 3.6GHz, 16GB RAM at 1867 MHz, and Intel Iris Graphics 540 1536 MB, running on battery power. The workload is centralised on the CPU, as shown in performance monitor in Figure 11, and the process took around 90-100% of the CPU time.

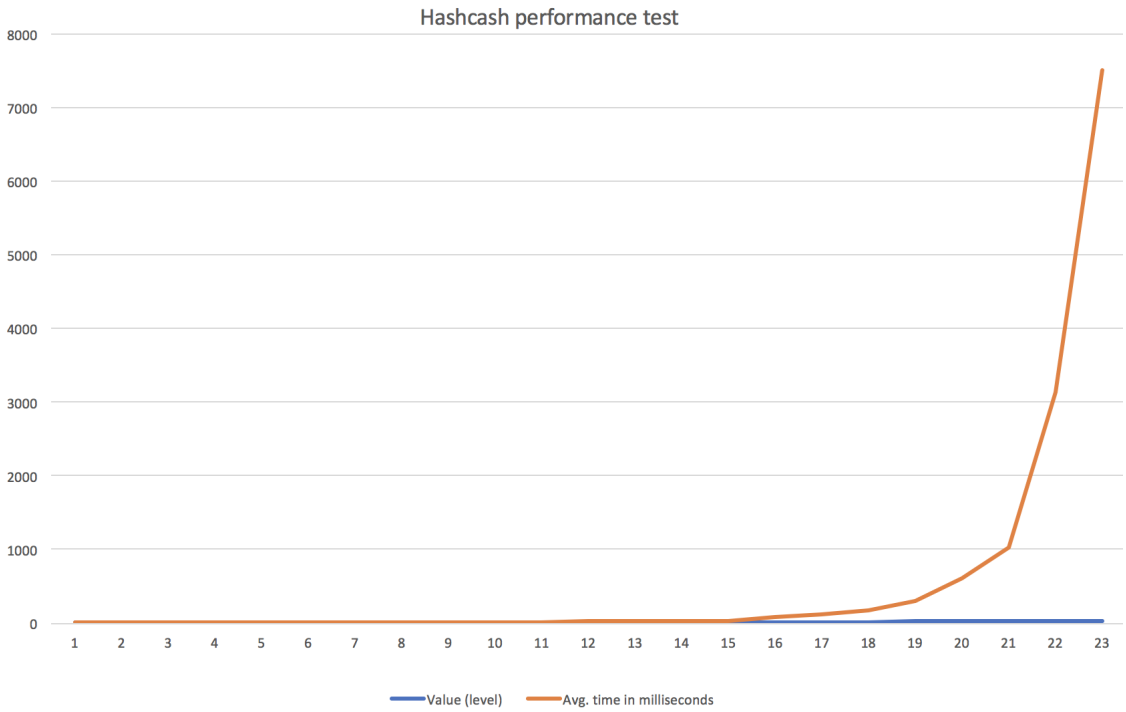
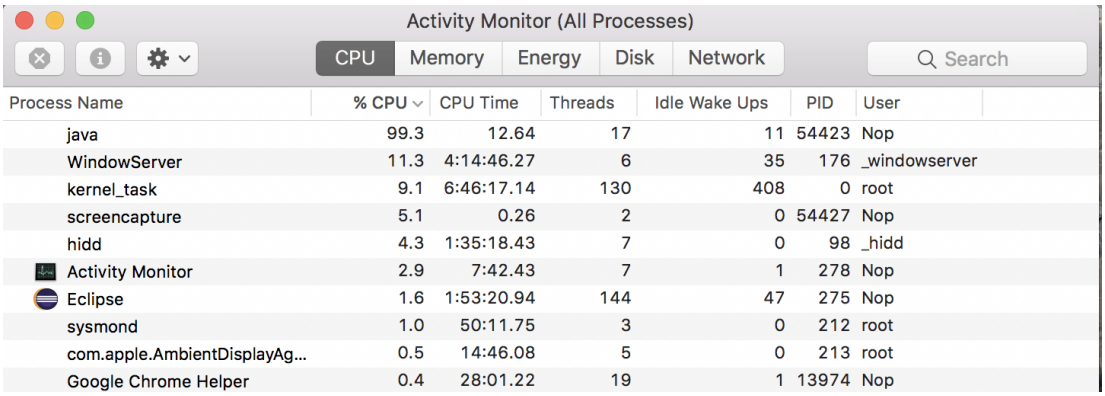


Figure 10: Value of Hashcash plotted against the average production time in milliseconds



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
java	99.3	12.64	17	11	54423	Nop
WindowServer	11.3	4:14:46.27	6	35	176	_windowserver
kernel_task	9.1	6:46:17.14	130	408	0	root
screencapture	5.1	0.26	2	0	54427	Nop
hidd	4.3	1:35:18.43	7	0	98	_hidd
Activity Monitor	2.9	7:42.43	7	1	278	Nop
Eclipse	1.6	1:53:20.94	144	47	275	Nop
sysmond	1.0	50:11.75	3	0	212	root
com.apple.AmbientDisplayAg...	0.5	14:46.08	5	0	213	root
Google Chrome Helper	0.4	28:01.22	19	1	13974	Nop

Figure 11: Performance monitor captured when running *mineKoins()* at level 20

With respect to the outcome of the performance test, we have decided to use 23 as the level of difficulty to slow down spammers from flooding the Blockchain. This allows a single Koin to be generated in roughly 7518 milliseconds on an average desktop computer like the one used for testing. A higher value could excessively slow down the starting phase of the system, while a slightly lower value will not be quite effective in slowing down spammers. This value should, however, be adjustable over time to suit the changing needs of the system. If the Blockchain is growing at a speed faster than ideal, the system should be able to adjust this to slow it down, and vice versa. This has been set as one of the core Future Work actions and we will examine the details later in the Conclusion and Future Work section.

4.5 Main.java

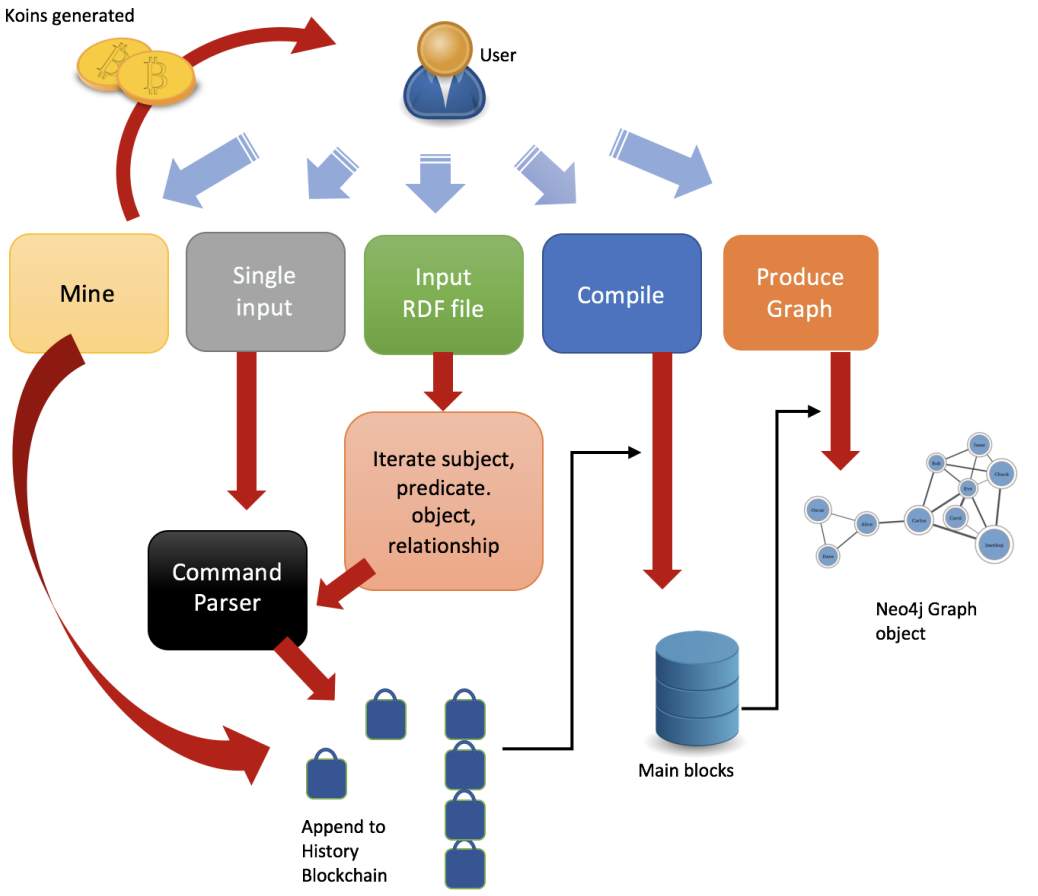


Figure 12: User's options when running the BIMS software locally

4.5.1 User Operations

For this prototype implementation, the user simply runs the program with no arguments and operates the system through the console. If there are existing serialised History Blockchain and main blocks in the local disk, they will be loaded and synchronised with the peers. The system then provides the user options that he/she can select to operate this system, including adding a single entry, adding an .rdf ontology file, running the Hashcash to produce some Koins, compiling the History Blockchain, and generating a Neo4j graph from the stored blockchain. These are summarised in Figure 12. Algorithm 2 explains how this is operated in a simplified pseudo code.

Algorithm 2 Main.java user operation loop

```
HistoryBC blockchain = loadHistoryBC_FromDisk().fetchUpdate()
BCCompiler compiler = loadBCCompiler_FromDisk()
int currentKoins = loadUserKoins()
boolean compiled = false
while true do
    operation = get user operation from console
    read(operation)
    if "mine" then
        //run PoW and add to History Blockchain
        Hashcash.mineKoins(blockchain)
        currentKoins++
    end if
    if match "[ADD/SIGN/REMV] [String[] or String] [int index] [int koins]" then
        blockchain.add(CommandParser.parse(operation))
    end if
    if match "RDF filename.rdf" then
        model = parse(filename) //parse ontology file using RDF4J
        for each Statement s in model do
            blockchain.add(CommandParser.parse("ADD" + s.subject))
            blockchain.add(CommandParser.parse("ADD" + s.predicate))
            blockchain.add(CommandParser.parse("ADD" + s.object))
            blockchain.add(Relationship(s.subject, s.predicate,s.object))
        end for
    end if
    if "compile" then
        compiled = true
        compiler.compile(blockchain)
        broadcast_update(blockchain)
    end if
    if "graph" then
        if compiled then
            compiler.generateGraph()
            break
        else
            exit("History Blockchain has to be compiled first")
        end if
    end if
    update_with_disk(blockchain)
end while
```

4.5.2 The Command Parser

The *CommandParser* class provides a static helper function parse() to the main loop. It takes a String input and constructs a *HistoryEntry* object accordingly. The command parser is only used to construct ADD, SIGN, and REMV entries; *parse*() subsequently invokes its sub-functions, *parseAdd*() and *parseSignRemv*() according to the Command detected in the given String. Since the procedures of SIGN and REMV are similar, they share the same constructing function.

```
private static HistoryEntry parseSignRemv(Command c, String[] split, PrivateKey priv, PublicKey pub) {
    if (split.length != 3) throw new IllegalArgumentException("Invalid no of argument for SIGN, REMV");
    int ref = -1;
    int koin = -1;
    try {
        ref = convertRef(split[1]);
        koin = Integer.parseInt(split[2].trim());
    } catch (Exception e) {
        e.printStackTrace();
    }

    return new HistoryEntry(c, ref, koin, priv, pub);
}


private static HistoryEntry parseAdd(String[] split, PrivateKey priv, PublicKey pub) {
    int koin = -1;
    Data d = null;
    try {
        koin = Integer.parseInt(split[split.length-1].trim());
    } catch (Exception e) {
        e.printStackTrace();
    }

    if(split.length > 3) {
        d = new Data(java.util.Arrays.copyOfRange(split,1,split.length-1));
    } else {
        d = new Data(split[1]);
    }
    return new HistoryEntry(d, koin, priv, pub);
}
```

4.5.3 Parsing an Ontology with RDF4J

As stated in Section 2.5.2, we utilize the RDF4J Java library in order to read an ontology and process it in our Java program. A *Rio* object provides static methods for parsing and writing RDF files. We parse the ontology file using the method *Rio*.parse() which takes the .rdf file as an *InputStream* and serialises its format – in our case *RDFFormat*.RDFXML, forming a *Model*. An RDF *Model* holds a set of RDF Statements to be iterated through. We can then repeatedly extract the subject, predicate and object from each Statement using *getSubject*(), *getPredicate*() and *getObject*() respectively.

4.6 Tracker Server and Networking

4.6.1 The Tracker Server

As stated, the system initially uses a tracker that allows new participants to join the network. The Tracker class implements this functionality. Tracker runs a server socket, listening on port 4444 – hardcoded for the time being. When a client connects to the tracker, the system forwards to the client the list "nodeaddresses" containing the IP addresses and the port numbers of all the participants in the network. Now, the client can use this list to connect to other peers in the network, asking for a copy of the History Blockchain. The main method of the Tracker class is shown in Appendix B Figure 16

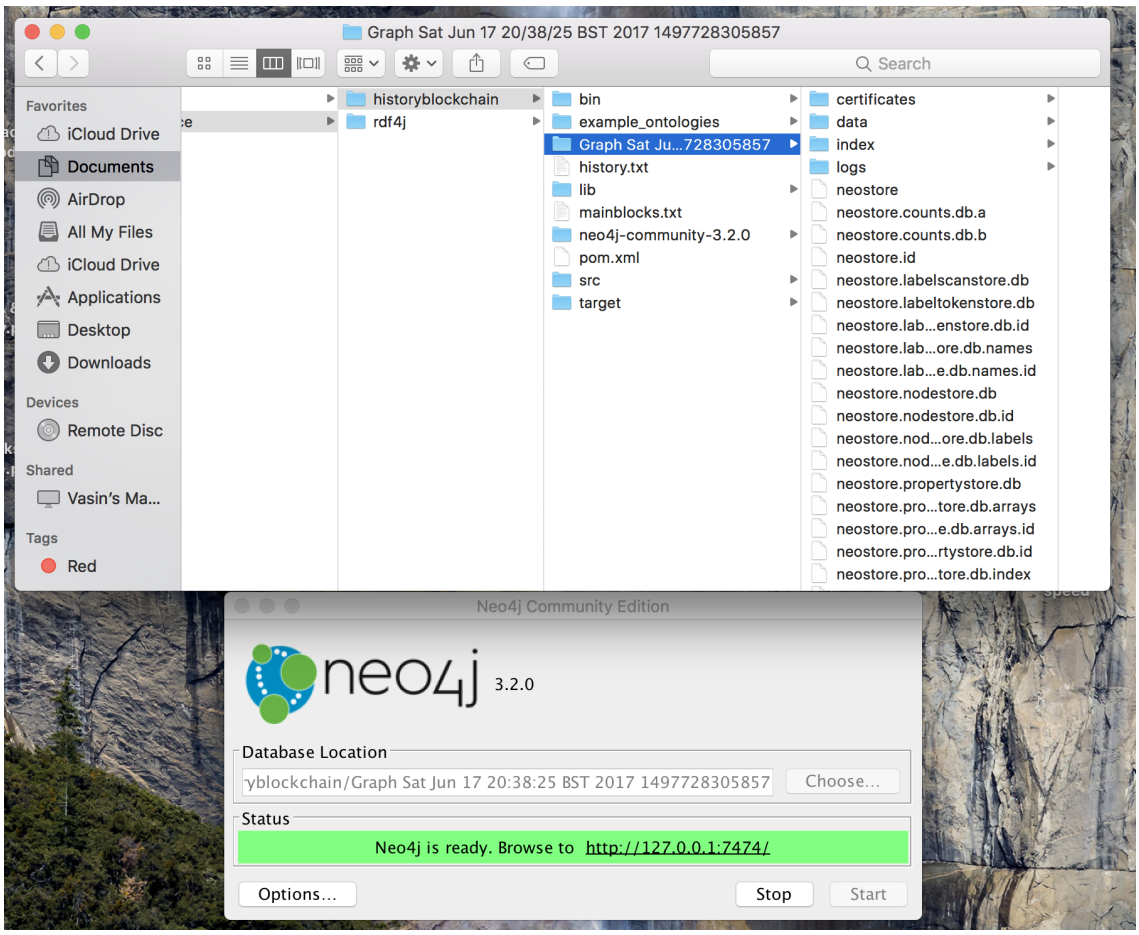
4.6.2 P2P Networking

Each participating node operates as a server, listening for new participants and asking for a copy of the History Blockchain or any broadcasted updates in the History Blockchain. Nodes also act as clients, sending any transactions they have appended to the peers in the network. The *send*()(Appendix B Figure 17) function in the *Broadcaster* class is invoked upon a complete compilation of the History Blockchain. If there are any new transaction(s) added by the user, these transactions get serialised and transmitted to the peers via a maintained list send to a separate port/server, solely reserved for recording updates. Therefore, each participating node runs two separate servers/ports: an update server for the new transactions, as well as a 'welcome' server, which allows new participants to receive a copy of the History Blockchain. The class $NodeServer$ spawns two threads to run these servers when the BIMS user software starts off, using $runWelcomeServer()$ and $runUpdateServer()$. The implementations can be found in the Appendix Section B Figure 18 and 19.

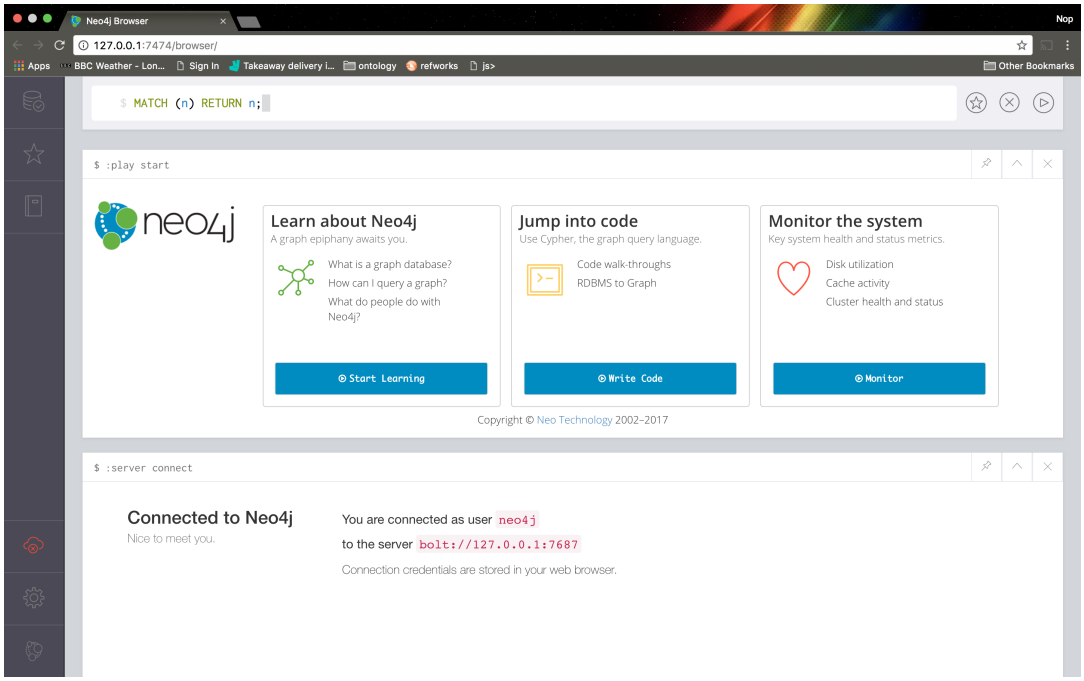
$HistoryBC$ HistoryBC implements the function $fetchUpdate()$ which is invoked before any user operations. Since the 'update' server saves transaction updates in files, sorted in chronological order, $fetchUpdate()$ retrieves all these files, deserialises them into a $List<HistoryEntry>$ object, and appends those transactions into the local History Blockchain. The implementation is demonstrated in Appendix Section B Figure 20. .

5 Testing

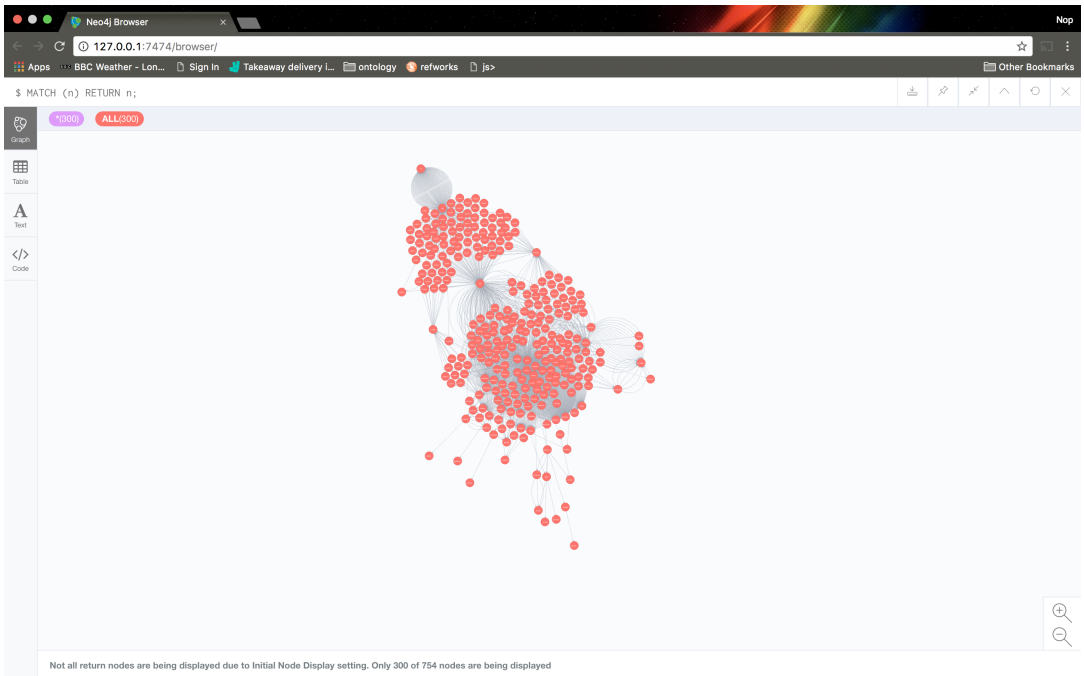
In this section we examine the resulting graph produced by BIMS after adding some sample ontology files into the system. After the insertion, we can then query entries in the graph using Neo4j's specialised query language *cypher*[37] through its Web interface. This test will involve the wine ontology(*wine.rdf*[38]) and the food ontology(*food.rdf*[39]).



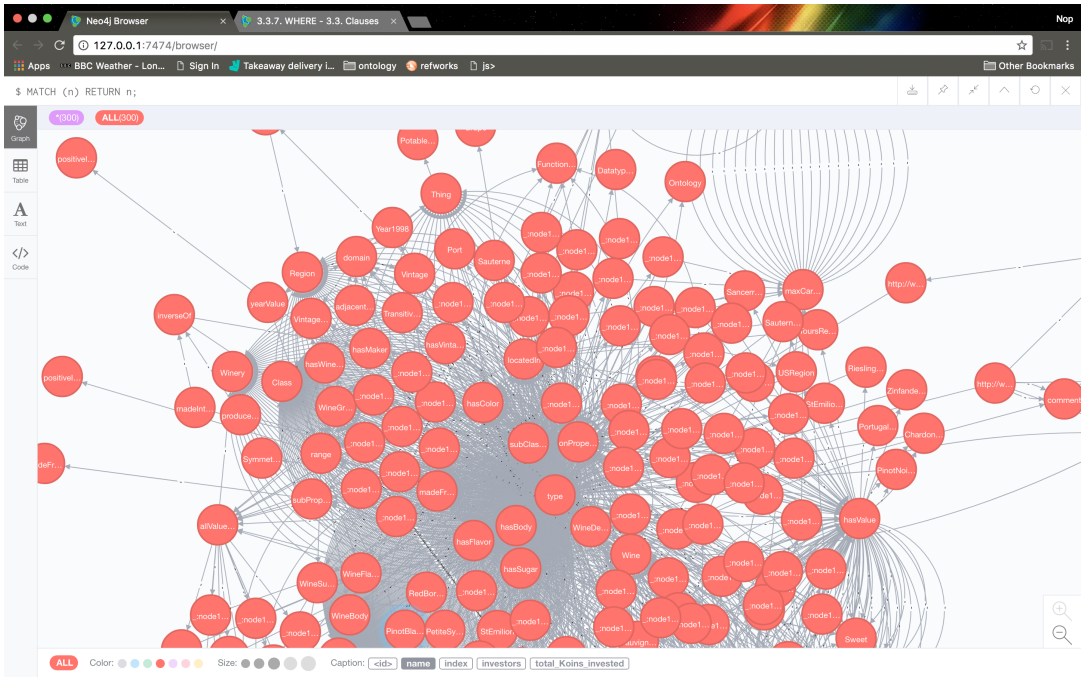
A Neo4j graph directory will be created upon a successful graph generation command. To view the graph, we run Neo4j and select the directory that has been created, then click "Start" and browse to the address indicated by the application.



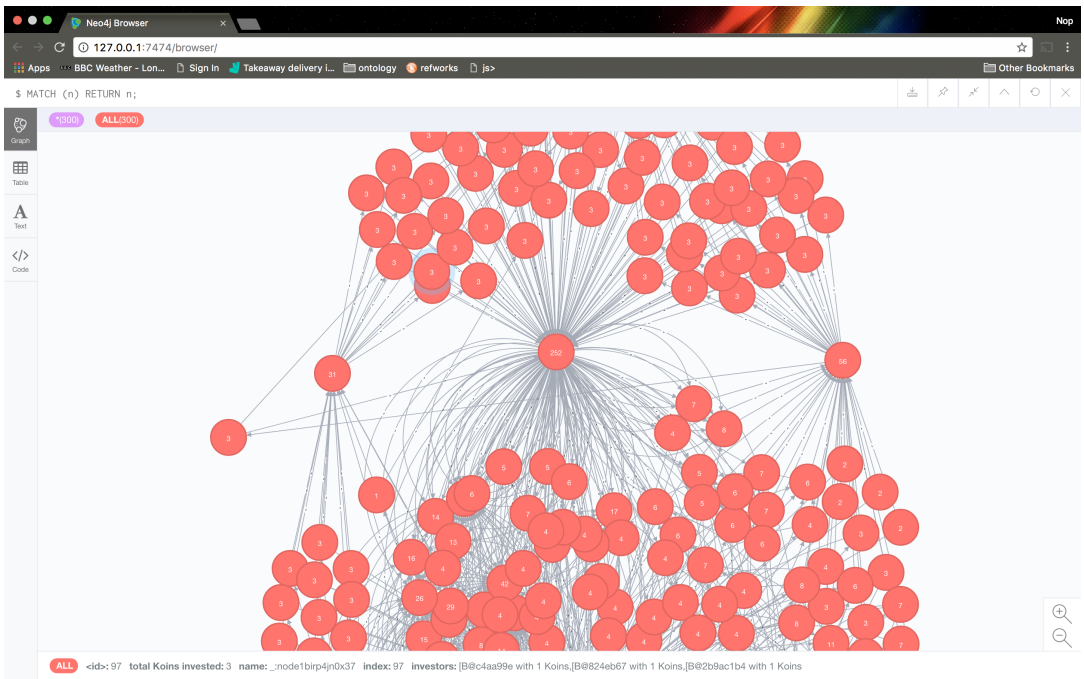
The user will be presented with a web Web user interface. We will use the cypher language to query data and visualise the contents of the compiled database. For example, "MATCH (n) RETURN n;" will return every nodes and their relationships in the graph.



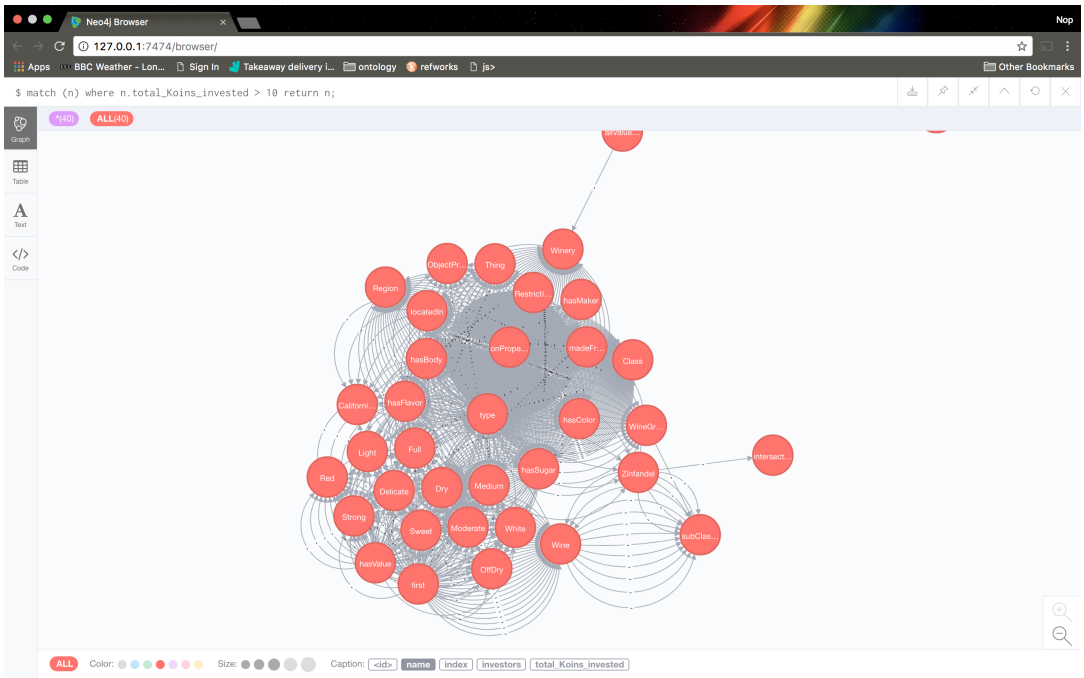
Here is an example of a graph database produced by compiling a blockchain for which only contained the wine ontology. Initially, Neo4j limits the number of displayed nodes to 300, hence it does not show all nodes here for rendering performance purposes. This setting can be changed with *:config initialNodeDisplay: [no. of display nodes]* and users can also look at the returned queried results in forms of tables or text if they choose to.



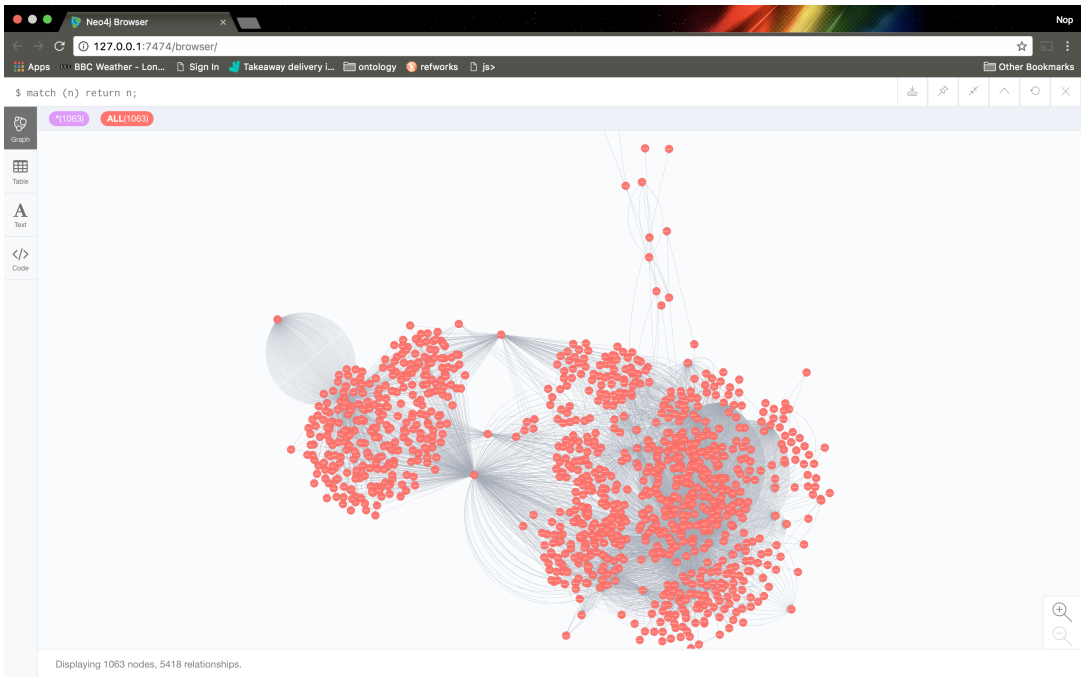
On the top left corner we can see the labels of every nodes in the graph, . since Since we used ALL for every node (as mentioned in Section 4.3.3), there is only 1 one label. We can select this label and choose how we want to display the nodes with for that label. For this instance, we are displaying the 'name' property of the nodes.



We can also look at other properties like the total Koins invested in each node. By selecting a node, we can observe all the property properties it holds at the bottom. ; e.g. the list of investors and their investments.



Cypher has a very powerful syntax for querying nodes and relationships. Here is an example of how we can view only nodes that have more than 10 total Koins invested and their relationships between each other



Here is the state of the graph after adding the food ontology into the system with sufficiently high limit of number of displaying nodes

6 Evaluation

In this section we will evaluate the result of the project and analyse the system in both qualitative(functionalities) and quantitative(performance) aspects. We will also discuss the strengths and weaknesses, what has been achieved and also what is lacking in the system.

6.1 Functionality

We have achieved a working prototype with minimal functionality. There are many components of the system which is are still not polished enough to allow for a releasable prototype production level system for able to support a large number of users. Some of which of these components are outlined below.

- KeyPair generation currently produces 512-bit keysize, which is sufficient for a prototype but not for a large scale deployment. We also need a better more advanced random number generator algorithm – definitely better than SHA1 which has was been recently cracked[40]. The system also needs a way to memorise the keys that have been already been generated to prevent the unlikely event of collisions.

- Serialisation is used to store the History Blockchain and the main blocks locally. This is done by the Java interface Serializable with a default serial version UID. There are 2 two major problems with this approach:

  1. The file produced in the disk is currently an unsecured text file which can be tampered with directly on the disk. This needs to be secured by some means.
  2. There could possibly be a more efficient way to store the History Blockchain and the main blocks which we have not explored. This could potentially reduce the cost of storage.

- BIMS needs a more polished user interface; possibly a Web interface to improve the ease-of-use. It should also allow users to avoid storing a full copy of the History Blockchain to participate participate even without storing a full copy of the History Blockchain. More details of on this will follow in the Future Work section.

6.1.1 Specification Comparison

We have outlined the specification of this system at the start of the project. Even though we moderately changed the final approach and a also made few major adjustments, we will now take a look back and compare the final product to the original design. The specification states that:

The system WILL offer the following:

1. *the ability to retrieve semantic data from a valid given Uniform Resource Identifier (URI)*

   We are no longer merely adding URIs and links to data in via the ontology management system. Our "adding an ontology" is now parsing a file with in RDF format into the system, instead of providing a locator for it, to simplify parsing and to focus on the Blockchain management system.

2. *a customised structure of a blockchain as means of securely storing encrypted/ hashed/ signed data as unique entities*

   This has been achieved, although initially the plan was not to create a History Blockchain but a database itself as a Blockchain. The History Blockchain has a customised Blockchain structure and inherits its immutability and integrity.

3. *digital signature for publishers to confirm the ownership of their data, so that others can trust who said what*

   Yes, a digital signature is required in order to append a 'transaction', e.g. adding data, into the History Blockchain.

4. *processing of RDF semantic models in order to classify data with respect to its taxonomy using* ***Apache Jena***

   The system processes and parses RDF semantic models with RDF4J instead of Apache Jena as described in Section 4.5.3. Although Apache Jena and RDF4J are very similar, RDF4J is more compatible with Ontotext's GraphDB which was our first choice in generating a graph database. For this reason, we switched to RDF4J after we decided to opt for Neo4j instead of Ontotext's GraphDB.

5. *ability to let users sign each entity as a mean of validating the information represented*

   Yes, users are free to 'invest' in entries should they agree with the data it holds.

6. *should the decentralized approach be chosen, the system will select and store data within chosen nodes of the system and enforce the 'health'(e.g. signed by a lot of users). A concept of BitTorrent.*

   BIMS is designed to be decentralised. All the participants, for now, owns a copy of the entire History Blockchain by using a P2P network approach similar to BitTorrent. As one of the possible future Future Work suggestions,

the system could perhaps allow partial node storing only for important/high valued parts of the History Blockchain.

7. *link newly added data with existing, correlated, and preferably healthy, data in the system according to their semantics*

   Correlated data are linked automatically to each other with relationships if they are parsed as an ontology from an .rdf file. It does not matter if the data they are linked with are 'healthy' (i.e. have a lot of investments) or not. The more often a particular entity is used in ontologies parsed, the more healthy and valuable it becomes since the prototype system requires user to invest – at least – 1 Koin in each entity added from an RDF file.

8. *presentation in a human-readable form by using graph database* ***Ontotext GraphDB***

   Yes, however we have switched the graph database service from Ontotext GraphDB to Neo4j due to its well-documented and easier to use API, as well as its powerful query engine.

The system SHOULD offer the following:

1. *ability to let users sign relationships between entities as a mean of validating the information represented, hence signing the 'concept' rather than objects*

   Users have the ability to sign the predicate acting as the relationship between data. They also have the ability to sign blocks containing series of data, rather than a single, atomic, data node.

2. *constant checking of the validity of the external data links should we decide not to store all data in a backup server*

   Since we are no longer working with external links, this is irrelevant.

3. *ability to let users report change in validity of data link, or change in the data themselves*

   Irrelevant due to above reason.

4. *breaking down chunks of data (e.g.articles) into atomic entries and establish the connection between each entry*

   Users can input Ontologies in the .rdf file format and they will be parsed and broken down to atomic entries, and also relationships will be created between them as defined in the RDF file. In order to feed plain text input, like, for example, an article, the system will require Natural Language Processing which

is out of scope for this project, as will be discussed in the Conclusions and Future Work section.

6.1.2 Comparison with Similar Systems

Steem

As stated in section 2.4.5, Steem is the closest existing system to BIMS, with the purpose to create a 'community' for people who value content. Steem focuses on social sites like Reddit, with its data stored on a blockchain-based database instead of a traditional server-base. Similar to BIMS, Steem participants can 'upvote' (i.e. invest) in content they find valuable. The users whose content is popular gets rewarded by credits. The content which is popular shifts the focus from the nonsensical entries added into the system – just like BIMS. Although BIMS does not reward owners of popular content, it is reasonable to do so in the future in order to create incentive for users to share valuable data.

Despite being similar in concept, there are a few major differences between BIMS and Steem. BIMS, rather than focusing on a social news platform, aims to prove 'the bigger picture' of the concept that a Blockchain structure can be beneficial to our traditional data management systems. BIMS is not restricted to semantic ontologies: users can take the algorithms and implementation defined in this project and adopt the system in a way they find useful. In short, BIMS can be seen as a kind of shared database which inherits 'trust' and 'integrity' by employing a Blockchain, and Steem as a social platform based on a Blockchain structure.

Nevertheless, Steem, like any other Blockchain-based system, shares the unresolved problem of scalability in Blockchains. The Blockchain grows very rapidly and gets very large in size. Since Steem currently has not addressed this issue in any way, a lot of Steem users avoid this problem by using it a 'service' rather than joining as a full node, and then store the entire Blockchain locally. Our system does not have capabilities to run partial nodes or to provide a similar service yet. We will discuss the scalability problem in more depth in the Conclusion section.

Steem has a market cap in cryptocurrency of over $546 million[24] at the time of writing this report, sitting at number 14 in the world's CryptoCurrency market capitalisation. The figure suggests that there is high interest in Steem in the market, and the approach that Steem is taking is being widely accepted. Hence we can expect the 'ideal' BIMS to attract an amount of attention from potential users.

BigchainDB

A distributed database which has been enhanced by Blockchain characteristics, BigchainDB shares resemblance with our system. BigchainDB's architecture is based on the InterPlanetary File System, IPFS with additional Blockchain features, promising immutability, high throughput, and low latency for its users. BIMS also

aims to inherit the immutability and the value of trust from the Blockchain structure, to improve the concept of a conventional database.

In contrast to BIMS, BigchainDB is advertised mainly for organisational and industrial uses. It allows a user to deploy a customised Blockchain database for personal or industrial use. BigchainDB does not require, or run, a credits system like Steem or BIMS. BIMS seeks to create a community-driven database where data can benefit anyone publicly, even though, as mentioned before, anyone can adopt BIMS with adjustments and use it in whatever way they consider beneficial.

However, once again, even though BigchainDB boasts an amazing read and write performance, it cannot avoid suffering from its Blockchain swiftly growing in size. Since it is built on top of RethinkDB, a real-time distributed JSON database, BigchainDB inherits all of its limitations, like, for example, memory performance limitations. Some analyst see this as a weakness in its security; RethinkDB can be the main target of attack to bring down BigchainDB[41].

BigchainDB claims to be trusted by the world's leading companies. They have been working in a wide range of areas, including automotive, energy, identity, IoT, and intellectual property with the likes of Microsoft, Toyota and Porsche. This is a promising sign for potential future interests in Blockchain-based database systems like ours.

6.2 Performance

We will be using three well-formed ontologies downloaded from two different sites in performance testing: *wine.rdf*[38], *food.rdf*[39], and *pizza.rdf*[42]. All test is carried out on a MacBook Pro on macOS Sierra version 10.12.2 with 2.3GHz dual-core Intel Core i5, Turbo Boost up to 3.6GHz, 16 GB RAM at 1867 MHz, and Intel Iris Graphics 540 1536 MB. We believe that the specs reflects an 'everyday' laptop for regular users.

6.2.1 Size and Scalability

Since scalability is a major problem in any Blockchain-based system, we will compare the size of the input ontology .rdf file(s) to the size of the History Blockchain produced, and see how it scales. We would expect the correlation to be linear, e.g. if adding an ontology with 300 "things" produced 1000KB Blockchain, adding one with 3000 "things" should produce a roughly 10000KB Blockchain

The smallest ontology tested, *pizza.rdf* , with 1251 entries, produced a 187 KB History Blockchain, while *food.rdf* with 2610 entries produced 385 KB and *wine.rdf* with 5517 entries produced 810 KB. Putting two or more ontologies simply results in increasing the size as expected, with all three ontologies producing 1381KB. Therefore, 1 KB in the History Blockchain can hold approximately 6.8 entries. Steem has the Blockchain size of 1.1 GB at July 15 2016[43]. Although this cannot be directly

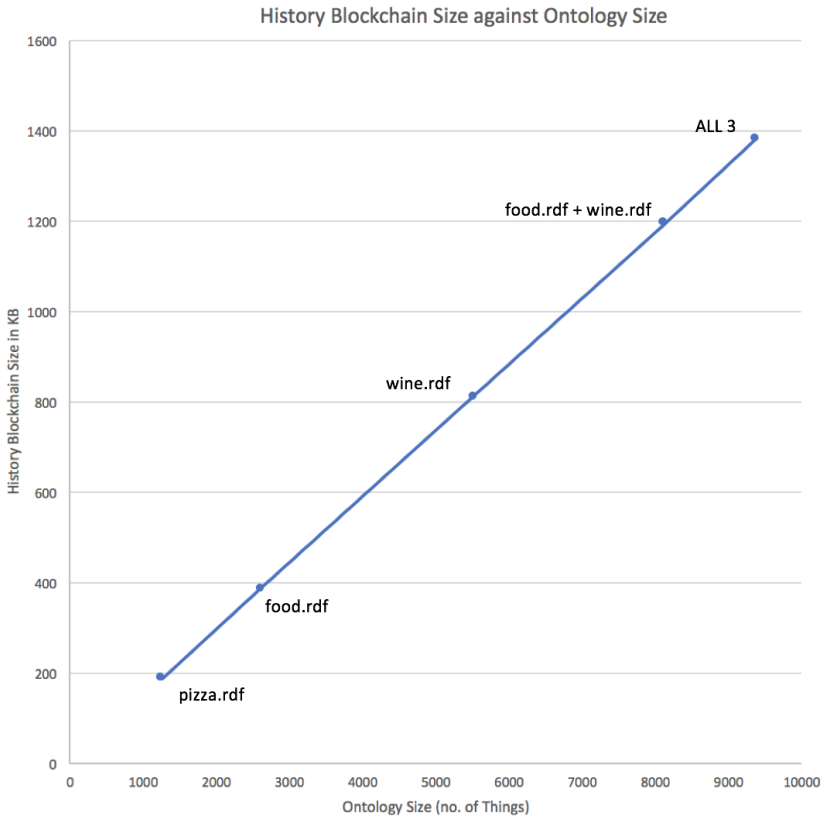


Figure 13: An plot of the size of History Blockchain produced against the size of input ontologies

compared because their Blockchain structure is different, we can state that if we were to distribute 1.1 GB of our Blockchain, there will be roughly 6,800,000 entries residing in it. If we treat the food ontology as average-sized, a Blockchain of this size could potentially hold just over 2600 ontologies, which is a notable amount. We believe that further optimisations, e.g. better serialisation, are necessary to reduce the size per entry and hence the overall size of the Blockchain.

6.2.2 Speed

In section 4.4.2 we have evaluated how Hashcash performed and decided on the appropriate level for BIMS. Now we will consecutively run the program from the start and mine just enough Koins before adding an ontology file to it, in order to see how long this process will take. Since we expect a Koin to be mined at $\tilde{7}.5$ seconds and since we consider the time for adding the entry as negligible (compared to the mining), the correlation between the size (in entries) of the ontology added and the time required should be linear

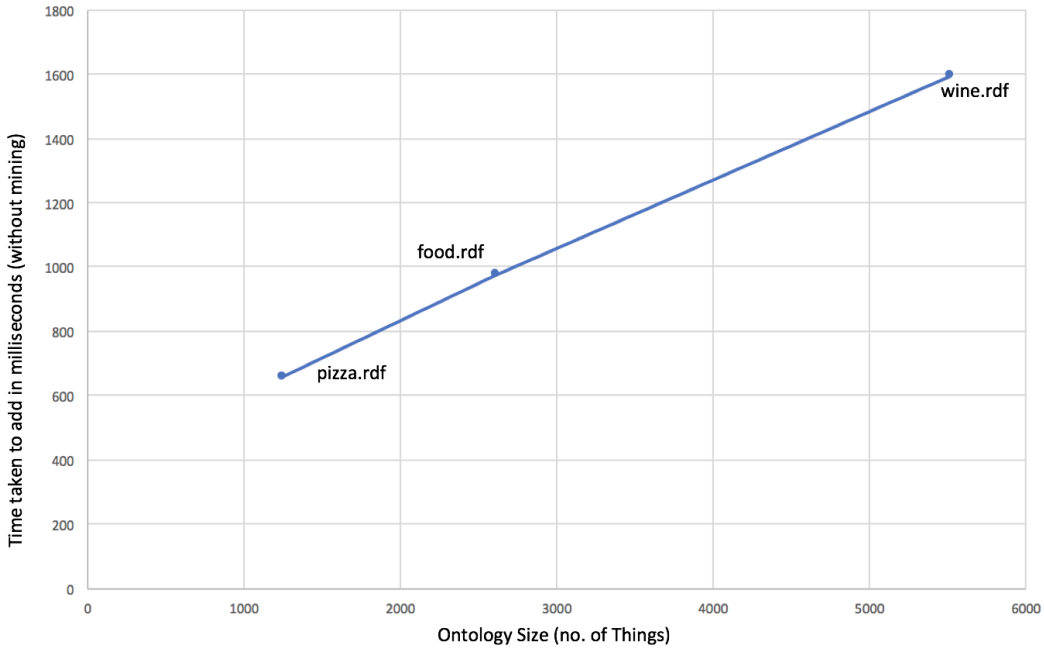


Figure 14: An plot of the time taken to add an ontology without mining against the size of input ontologies

Mining 1251 Koins for 1251 entries in *pizza.rdf* and adding them into the system took 9,315,605 milliseconds which is approximately 2.5 hours, with the adding process taking only 659 milliseconds of that sum. So, the average mining rate is (9,315,605 - 659)/1251 = 7446 milliseconds per Koin, which is close to the average observed in section 4.4.2. Since we have proved that this rate is the approximate average rate of 1251 runs, we can use this average and estimate only the time required to add the entries from other ontologies. *food.rdf* will need 977 milliseconds and should approximately take 7446*2610 + 977 = 19,435,037 milliseconds ($\tilde{5}$.4 hours) for the whole process, while *wine.rdf* will need 1594 milliseconds for the adding process and will roughly take 7446*5517 + 1594 = 41,081,176 milliseconds ($\tilde{1}$1.4) hours for the entire process. This establishes the 'inefficiency' we are looking for! As mentioned in Section 4.4.2, the speed of the mining process can be made adjustable by shifting the level of the Hashcash algorithm in order to control the system's growth rate.

6.3 Strengths and Weaknesses

To sum up the evaluation of our system, we will look at some of the strengths and weaknesses the system possesses. Some of the weaknesses have already been introduced above and we will review some proposals in resolving them in the Conclusion and Future Work section.

Strengths

- We have achieved a new type of distributed database where all manipulations (i.e. transactions) are recorded and can be verified. This builds a *trust layer* for data and guarantees integrity.
- The system operates in a fully decentralised manner, meaning there is no single point of failure.
- BIMS inherits immutability from the Blockchain architecture. It is computationally exhausting for an attacker to change or revert any block in the History Blockchain.
- The system operates with a flexible level of **inefficiency** which is one of the core strengths of using a Blockchain-based structure. This can be adjusted according to the rate the Blockchain is growing.

Weaknesses

- As with any other Blockchain-based systems, the size of the Blockchain can grow very rapidly. This increases the storage required for participants and present the problem of scalability. We will discuss the issue in more details in Section 7.3.
- There is currently no friendly user interface other than Neo4j's engine, so ease-of-use is lacking.
- A participant is currently required to run a full node by storing a copy of the entire History Blockchain. When this gets larger, it demotivates a potential user to join the network.
- Users do not get rewarded for useful interactions with the system, e.g. for adding and investing valuable content, and there is no real monetary value in the Koins. Unless a user really cares about the "economy of information", there is no incentive for one to participate. However, we have showed that similar systems are already popular and noteworthy.
- The Hashcash algorithm currently requires miners to compute 'useless' hashes, while this computational power could be used in a more constructive manner. We will touch upon some ideas in the Future Work section.

- The Hashcash algorithm is centralised on CPU power. This means participants can get specialised hardware for mining – a strong processing unit in this instance, and potentially 'dominate' the system, as has happened with Bitcoin Mines[44].
- A participant has no way to check if their local copy has been tampered with, apart from running all transactions in the History Blockchain from time zero. This is burdensome and computationally demanding. There is, however, a solution for this which will be introduced in the next section.
- It is difficult to compare the state of History Blockchain without compiling since transactions can be in different order due to the way P2P broadcast system operates.

7 Conclusion and Future Work

As the project comes to an end, we will shortly summarise the achievements of this project and list some possible future work and extensions that can be taken to refine BIMS in different ways.

7.1 Conclusion

We have succeeded in adopting a Blockchain structure to create an integrity-guaranteed distributed database with a trust layer. Despite the fact that there currently are numerous weak points in the system and thus the prototype requires a great deal of polishing to be market-ready, we have established the direct benefits that the Blockchain architecture has compared to a conventional database. We also discussed many systems in the market which fuse the two ideas together in similar and different approaches, suggesting that interest in the concept already exists and is unceasingly growing. Perhaps we could be expecting a similar system to leave a significant mark to the ever-growing world of technologies in the near future.

As for BIMS, despite the long list of weaknesses it appears to have, we are pleased with the results we accomplished. In the future, if extensions like the Natural Language Processing (discussed below) can be achieved, we estimate that the system can be very influential to the world of information. BIMS can be employed as the core of systems focused on contributing to the open information domain like Metakosmos, by providing the *trust* they tend to lack. Currency has been playing a big role in the modern world, but so has information. This work was done with the belief that information should be the most powerful influence to currency – not the other way around, and should continue to be in the future.

7.2 Future Work

There are potentially countless improvements to be made. Here we will list some major ones which could allow this prototype to reach production level quality in the future.

7.2.1 Natural Language Processing

BIMS is currently mainly focused on ontology inputs, but in the future we can also incorporate Natural Language Processing and use non-ontology input such as books and articles as input for the system. NPL should then process and create relationships according to their semantics and then insert the entities into the History Blockchain.

7.2.2 State-of-the-Blockchain Hash

As mentioned previously, a participant has no way to check if their local copy has been tampered with, apart from running all transactions in the History Blockchain from time zero. This can be solved by periodically adding a block containing the *hash of the History Blockchain state* into the Blockchain every X blocks or every X hours. With this supplementary block, users can get a hash of the state of their local copy, compare the two hashes and verify that their local copy has not been tampered with much faster than having to re-calculate the entire blockchain.

Such a block can also be used to 'update' (i.e. alter) the requirements for appending to the Blockchain. For example, it can state the new minimum fee for making a transaction (currently 1 Koin), in order to adapt to the 'value' of the Koins at that stage. The block can also possibly specify a different Proof-of-Work algorithm for the economy, or set a new level of difficulty for Hashcash, in order to slow down or speed up the system's rate of growth. Bitcoin uses a similar strategy: the target value of a valid block hash is changed every 2016 blocks so that a block is mined at the desired rate of the system (approx. 1 block every 10 minutes)[45].

7.2.3 Information Stock Exchange

BIMS' credits system, Koins, does not hold any monetary value, nor reward participants for good behaviour in the system. This gives users no incentive to take part actively. BIMS can be made to reward users who add and invest in good content (like Steem) as a motivation for participants. The system can potentially afford this since it is collecting fees from all instructions and transactions made by its users. In addition to this, Koins can also be made to hold financial value and become free-to-trade. This is expected to increase user interest in this system, as has happened with other similar systems mentioned in Section 2, and could potentially allow our system to become self-sustainable, by using some of the Koins it collects (from its transaction fees) to pay for server and network costs.

7.2.4 Constructive Proof-of-Work Algorithm

As stated in the previous section, Hashcash currently computes useless hashes to mine Koins. This processing power, rather than going to waste, can be used in more productive ways. At the moment we cannot find a way to compute a useful process to benefit BIMS, but there are numerous distributed computing projects[46] which ask for donation in computing power. For instance, the DreamLab project uses computational power to run a Cancer Research project.

The choice of proof-of-work algorithm can also be changed, since Hashcash is centralised on the CPU. If this mechanism was not available, it could allow users with specialised mining hardware to control the majority of the power in the system, which is not necessarily good for such systems since our aim is for information to

control the currency – not the converse. There are other algorithms, like, for instance the CryptoNight hash function[47], that are based on other components like GPU, memory, cache, etc.

7.2.5 User Interface and Running Partial Nodes

The system needs a polished, special purpose, user interface which would allow to easily query the current compiled version of the database, without having to learn how to express themselves in RDF4J. Users should also be able to get only parts of the History Blockchain instead of a full copy when it grows larger. There could be full nodes (i.e. computers with the entire blockchain) running Web services allowing users to participate and create transactions without having to become a node themselves. Steem also provides such service to avoid the scalability problem; however, it doesn't solve the issue and defeats the purpose of the system being decentralised. If a lot of clients chose to use the Web service rather than using the system as a node, the Web service is a potential candidate target for attackers.

7.2.6 Encryption and Private Network

All the data in BIMS are publicly available and unencrypted in order to benefit every user in the system. However, it could perhaps also be used with encrypted data instead. *Hawk*[48], a smart contract system based on the Blockchain, is capable of storing encrypted financial data, retaining privacy for users who wish to remain hidden. Encryption of data allows private groups to be formed: only those who know how to decrypt specific blocks can use the data. Hawk also allows users who do not have access to encrypted data to verify that certain encrypted data meet certain conditions without seeing the actual data, using a *zero knowledge proof.*[49].

Private networks and zero knowledge proofs can potentially be adopted by our system to allow for private groups to be formed. This opens up a wide range of possible use cases for our system; e.g. to be used as a core for privacy-preserving cryptocurrency, like Zero-cash[50].

7.3 The Scalability Problem

The biggest unresolved issue with everything involving a Blockchain is scalability. Even similar systems, like Steem, are not actively trying to solve the scalability problem. There are discussions on Moore's Law enabling us to continue to achieve the throughput we desire in the future as processing power and memory capacity growing consistently, along with the increasing network speeds. This, however, does not solve the problem, but merely prevents it from getting worse in the near future.

However, there is on-going work on a Scalable Blockchain Protocol using Proof-of-Stake[51] that is being attempted by the co-founder of Ethereum, Vitalik Buterin[52]. In a proof-of-stake algorithm for Ethereum, the miner of each block is not chosen by competition in processing power, but is instead selected from a pool. The choice is influenced by the total number of users' coins and the current complexity of the network. This makes proof-of-stake greener than proof of work, without having to compute hashes.

Even though, as Iain Stewart – a Bitcoin researcher at Imperial College – stated, Buterin's attempts are part of *"a very complicated paper and the logic does not fall through naturally"*, it is evidence that work in solving this issue is currently being carried out – even if it is making slow progress. At this moment, we can only hope that the problem will be resolved in the future, for the benefit of all Blockchain-based systems.

8 References

[1] T. B. Lee. (2017). Bitcoin's price keeps breaking records. here's what's driving its growth, [Online]. Available: https://www.vox.com/new-money/2017/5/26/15687062/bitcoin-bubble-explained.

[2] J. Bagley. (2016). What is blockchain technology, [Online]. Available: https://blockgeeks.com/guides/what-is-blockchain-technology/.

[3] A. M. Antonopoulos, *Mastering bitcoin*, First. Sebastopol, CA; Beijing: O'Reilly Media, Inc.; O'Reilly, 2014, Andreas M. Antonopoulos.; xxi, 272 pages : illustrations ; 24 cm; text; unmediated; volume; Includes index., ISBN: 9781449374044.

[4] G. Karame and E. Androulaki, *Bitcoin and blockchain technology*. Boston: Artech House, 2016; 2016, Ghassan Karame, Elli Androulaki.; xiii, 218 pages : illustrations (black and white) ; 24 cm; still image; text; unmediated; volume; Includes bibliographical references and index.; Artech House information security and privacy series., ISBN: 1630810134; 9781630810139.

[5] Wikipedia. (2016). Bitcoin block data, [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/7/7a/Bitcoin_Block_Data.png (visited on 08/02/2016).

[6] E. F. (Ethereum). (2016). Ethereum project, [Online]. Available: https://www.ethereum.org/.

[7] R. W. Group. (2008). Resource description framework (rdf), [Online]. Available: https://www.w3.org/rdf.

[8] W. W. W. C. W3c. (2008). Sparql query language fo rdf, [Online]. Available: https://www.w3.org/TR/rdf-sparql-query/.

[9] L. Yu and SpringerLink, *Developer's guide to the semantic web*, eng; English, 2014.

[10] O. Corporation. (2008). Distributed database, [Online]. Available: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch21.htm.

[11] C. SINGH. (2016). Acid properties in dbms, [Online]. Available: http://beginnersbook.com/2015/04/acid-properties-in-dbms/.

[12] G. Images. (2016). P2p and client server solution, [Online]. Available: https://www.google.co.uk/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=0ahUKEwidiKvDwoXSAhVIRhQKHfTcDUsQjhwIBQ&url=http%3A%2F%2Fhelalinden.com%2Farticle%2Fpaper-2016678076%2F&psig=AFQjCNEYR9W3SC24dIaHEY88
ust=1486815585876953.

[13] J. Cope. (2002). Peer-to-peer network, [Online]. Available: http://www.computerworld.com/article/2588287/networking/peer-to-peer-network.html.

[14] E. Rescorla. (2008). Introduction to distributed hash table, [Online]. Available: http://www.computerworld.com/article/2588287/networking/peer-to-peer-network.html.

[15] Topcoder. (2011). Greedy is good, [Online]. Available: https://www.topcoder.com/community/data-science/data-science-tutorials/greedy-is-good/.

[16] I. BitTorrent. (2016). Bittorrent, [Online]. Available: www.bittorrent.com/.

[17] E. V. der Sar. (2006). Bittorrent: The "one third of all internet traffic" myth, [Online]. Available: https://torrentfreak.com/bittorrent-the-one-third-of-all-internet-traffic-myth/.

[18] J. Benet, "Ipfs-content addressed, versioned, p2p file system," 2014.

[19] M. I. of Technology. (2016). The solid project, [Online]. Available: https://solid.mit.edu/#home (visited on 08/02/2016).

[20] M. English, S. Auer, and J. Domingue, "Block chain technologies & the semantic web: A framework for symbiotic development," 2016.

[21] K. Gkoutzis and V. Geroimenko, "Moving from folksonomies to taxonomies: Using the social web and 3d to build an unlimited semantic ontology," in *Information Visualisation (IV), 2011 15th International Conference on*, IEEE, 2011, pp. 205–210.

[22] K. Gkoutzis, "A semantic web based search engine with x3d visualisation of queries and results," 2013.

[23] D. L. N. S. V. Z. B. J. J. C. M. Vandeberg, "Steem, an incentivized, blockchain-based social media platform.," 2016.

[24] CoinMarketCap. (2017). Cryptocurrency market capitalization, [Online]. Available: https://coinmarketcap.com/.

[25] D. Irvine. (2017). Maidsafe, the new decentralized internet, [Online]. Available: https://maidsafe.net/.

[26] J. Zawistowski. (2017). Golem, [Online]. Available: https://golem.network//.

[27] T. M. R. M. A. M. D. D. J. T. T. M. G. M. R. H. S. B. A. Granzotto, "Bigchaindb: A scalable blockchain database," 2016.

[28] I. Neo Technology. (2017). Neo4j, the world's leading graph database - nro4j graph databse, [Online]. Available: https://neo4j.com/.

[29] O. Corporation. (2017). Ontotext graphdb, [Online]. Available: http://ontotext.com/products/graphdb/.

[30] E. RDF4J. (2017). Eclipse rdf4j, a java framework for rdf, [Online]. Available: https://neo4j.com/developer/java/.

[31] A. Back. (2003). The hashcash proof-of-work function, [Online]. Available: http://www.hashcash.org/.

[32] BitcoinWiki. (2017). Hashcash, [Online]. Available: https://en.bitcoin.it/wiki/Hashcash.

[33] J. HAWES. (2011). The bittorrent protocol, [Online]. Available: http://www.morehawes.co.uk/the-bittorrent-protocol.

[34] Oracle. (2017). Java.security (java se 7), [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html/.

[35] I. Neo Technology. (2017). Using neo4j from java, [Online]. Available: https://neo4j.com/.

[36] G. Rubin. (2006). Nettgryppa libraries, [Online]. Available: http://www.nettgryppa.com/code/HashCash.java.

[37] I. Neo Technology. (2017). Neo4j's graph query language: An introduction to cypher, [Online]. Available: https://neo4j.com/developer/cypher-query-language/.

[38] W3C. (2009). W3c web ontology language guide: Example wine.rdf, [Online]. Available: https://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf.

[39] ——, (2009). W3c web ontology language guide: Example food.rdf, [Online]. Available: https://www.w3.org/TR/2004/REC-owl-guide-20040210/food.rdf.

[40] M. S. E. B. P. K. A. A. Y. Markov, "The first collision for full sha-1," 2011.

[41] reddit mamber "killerstorm". (2016). Bigchaindb: A prime example of blockchain bullshit, [Online]. Available: https://www.reddit.com/r/Bitcoin/comments/4j7wjf/bigchaindb_a_prime_example_of_blockchain_bullshit/.

[42] T. U. of Manchester School of Computer Science. (2004). Examples/ pizza ontology, [Online]. Available: http://www.cs.man.ac.uk/~drummond/cs646/examples/pizzas2_0.owl.

[43] S. U. @kingkrung. (2016). Thread: Does anyone know the current blockchain size of steem? [Online]. Available: https://steemit.com/steem/@kingkrung/does-anyone-know-the-current-blockchain-size-of-steem.

[44] E. Franco. (2015). Inside the chinese bitcoin mine that's grossing $ 1.5m a month, [Online]. Available: https://motherboard.vice.com/en_us/article/chinas-biggest-secret-bitcoin-mine.

[45] B. Wiki. (2017). Difficulty, [Online]. Available: https://en.bitcoin.it/wiki/Difficulty.

[46] K. Pearson. (2017). Distributed computing - active projects, [Online]. Available: http://www.distributedcomputing.info/projects.html.

[47] M. J. T. N. N. A. M. Juarez, "Cryptonight hash function," 2013.

[48] A. K. A. M. E. S. Z. W. C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," 2015.

[49] M. Green. (2014). Zero knowledge proofs: An illustrated primer, [Online]. Available: https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer/.

[50] E. B.-S. A. C. C. G. M. G. I. M. E. Tromer. and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," 2014.

[51] BlockGeeks. (2017). Proof of work vs proof of stake: Basic mining guide, [Online]. Available: https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/.

[52] V. Buterin, "Notes on scalable blockchain protocols (version 0.3.2)," 2015.

9 Appendix

A. Glossary

BIMS: the name of the system, short for Blockchain-based Information Management System

Compiler: the software executing the algorithm to convert the *History Blockchain* to a local database and the *main blocks*

Hashcash: the proof-of-work algorithm used by the system during the mining process. It requires the miner to compute a hash of a specified string until certain conditions are met.

History Blockchain: the append-only Blockchain recording all *transactions* of the database as well as any *mining* activities

Koins: the credits/currency used for transactions in the system

Main Blocks: a defined series of storage objects formed by compiling the transactions in the *History Blockchain*

Mining: the process of generating Koins, i.e. the credits used for transactions

Tracker: a server which keeps track of the IP addresses of the active members of the network. New participants can request this list from a tracker to join the system.

Transaction: any operation that aims to edit manipulate the database in any way is considered to be a transaction.

B. Codes

```
public void genGraph() {
    File filename = new File("Graph "+ last_update + " " + last_update.getTime());
    GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
    GraphDatabaseService db = dbFactory.newEmbeddedDatabase(filename);
    try (Transaction tx = db.beginTx()) {

        for (MainBlockEntry entry : entries) {
            if (entry.getSecondData() != null) {

                //relationship entries case

                int first = search(entry.getData());
                int second = search(entry.getSecondData());

                if (first < 0 || second < 0 || first == second) {
                    System.out.println("invalid relationships at gengraph(), should never happen");
                    continue;
                }

                Node it = db.findNode(Label.label("ALL"), "name", entries.get(first).getData().toString());
                Node it2 = db.findNode(Label.label("ALL"), "name", entries.get(second).getData().toString());
                it.createRelationshipTo(it2, RelationshipType.withName("-"));



            } else {
            //create new node and set no. of koins as property for each entries
            Node node = db.createNode(Label.label("ALL"));
            node.setProperty("index", entry.getIndex());
            node.setProperty("name", entry.getData().toString());


                if (entry.getMiner() == null) {
                    //not a PoW node, carry on.......
                    node.setProperty("total Koins invested", entry.getTotalKoins());

                    //turn investments into array of string which is an allowed type of property
                    List<Inv> invs = (List<Inv>) entry.getInvestments();
                    String[] investors = new String[invs.size()];
                    int count = 0;
                    for (Inv i : invs) {
                        investors[count] = i.getPub() + " with " + i.getKoins() + " Koins";
                        count++;
                    }
                    node.setProperty("investors", investors);
                } else {
                    //PoW node-----------------------------------------
                    node.setProperty("PublicKey", entry.getMiner());
                    node.setProperty("STAMP", entry.getData().toString());
                }



            }
        }

        tx.success();

    }

}
```

Figure 15: *genGraph*() function in *BCCompiler* class

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

    ServerSocket serverSocket = null;
    Socket socket = null;
    OutputStream out = null;
    InputStream in = null;
    BufferedWriter writer = null;

        serverSocket = new ServerSocket(4444);

        while(true) {
            try {

                socket = serverSocket.accept();
                String IP = socket.getInetAddress().toString() + socket.getPort();
                File file = new File("nodeaddresses.txt");

                // Get the size of the file
                long length = file.length();

                if (length > Integer.MAX_VALUE) {
                    System.out.println("File is too large.");
                }

                byte[] bytes = new byte[(int) length];

                in = new FileInputStream(file);
                out = socket.getOutputStream();

                int count;

                while ((count = in.read(bytes)) > 0) {
                    out.write(bytes, 0, count);
                }

                //record this client IP to the file
                FileWriter fstream = new FileWriter("nodeaddresses.txt", true); //true = appends
                writer = new BufferedWriter(fstream);
                writer.write(IP);
                writer.newLine();

            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (writer != null) writer.close();
                if (out != null) out.close();
                if (in != null) in.close();
                if (socket != null) socket.close();
                if (serverSocket != null) serverSocket.close();
            }

        }

}
```

Figure 16: *Main*() method in *Tracker* class

```java
//send updates.txt in directory to others
public void send(String filename) {
    for (String p : peers) {
        //[0] = IP and [1] = port number
        String[] address = p.split(":");

        Socket socket = null;
        String host = address[0];
        int port = Integer.parseInt(address[1]);
        InputStream in = null;
        OutputStream out = null;

        try {
            socket = new Socket(host, port);

            File file = new File(filename);

            // Get the size of the file
            long length = file.length();

            if (length > Integer.MAX_VALUE) {
                System.out.println("File is too large.");
            }

            byte[] bytes = new byte[(int) length];

            in = new FileInputStream(file);
            out = socket.getOutputStream();

            int count;

            while ((count = in.read(bytes)) > 0) {
                out.write(bytes, 0, count);
            }


        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (out != null) out.close();
                if (in != null) in.close();
                if (socket != null) socket.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Figure 17: *send*() method broadcasts updates to every peers maintained in a list in *Broadcaster* class

```java
//start p2p node, this waits for anyone to connect and ask for a copy of the blockchain
    private static void runWelcomeServer() {
         (new Thread() {
                @Override
                public void run() {
                    ServerSocket serverSocket = null;
                    Socket socket = null;
                    OutputStream out = null;
                    InputStream in = null;
                    BufferedWriter writer = null;


                        while(true) {
                            try {

                                serverSocket = new ServerSocket(5555);
                                socket = serverSocket.accept();
                                String IP = socket.getInetAddress().toString() + socket.getPort();
                                File file = new File("history.txt");

                                // Get the size of the file
                                long length = file.length();

                                if (length > Integer.MAX_VALUE) {
                                    System.out.println("File is too large.");
                                }

                                byte[] bytes = new byte[(int) length];

                                in = new FileInputStream(file);
                                out = socket.getOutputStream();

                                int count;

                                while ((count = in.read(bytes)) > 0) {
                                    out.write(bytes, 0, count);
                                }

                                //record this client IP to the file
                                FileWriter fstream = new FileWriter("nodeaddresses.txt", true); //true = appends
                                writer = new BufferedWriter(fstream);
                                writer.write(IP);
                                writer.newLine();

                            } catch (Exception e) {
                                 e.printStackTrace();
                            } finally {
                                try {
                                    if (writer != null) writer.close();
                                    if (out != null) out.close();
                                    if (in != null) in.close();
                                    if (socket != null) socket.close();
                                    if (serverSocket != null) serverSocket.close();
                                } catch (Exception e) {
                                    e.printStackTrace();
                                }
                            }
                        }
                }
            }).start();
    }
```

Figure 18: *runWelcomeServer*() function in *NodeServer* class

```java
//start server for receiving updates for other peers
private void runUpdateServer() {
    (new Thread() {
        @Override
        public void run() {
            ServerSocket serverSocket = null;
            Socket socket = null;
            OutputStream out = null;
            InputStream in = null;


            while(true) {
                try {
                    serverSocket = new ServerSocket(6666);
                    socket = serverSocket.accept();

                    //save received update into file in chronological order
                    in = socket.getInputStream();
                    out = new FileOutputStream(new Date().getTime() + "updates.txt");
                    byte[] bytes = new byte[16*1024];

                    int count;
                    while ((count = in.read(bytes)) > 0) {
                        out.write(bytes, 0, count);
                    }


                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    try {
                        if (out != null) out.close();
                        if (in != null) in.close();
                        if (socket != null) socket.close();
                        if (serverSocket != null) serverSocket.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }).start();
}
```

Figure 19: *runUpdateServer*() function in *NodeServer* class

```java
public void fecthUpdate() {
    //get all update files in the directory
    File dir = new File(".");
    File [] files = dir.listFiles(new FilenameFilter() {
        @Override
        public boolean accept(File dir, String name) {
            return name.endsWith("updates.txt");
        }
    });

    //de-serialize and add append each one
    FileInputStream fi = null;
    ObjectInputStream oi = null;

    for (File f : files) {
        try {
            fi = new FileInputStream(f);
            oi = new ObjectInputStream(fi);

            @SuppressWarnings("unchecked")
            List<HistoryEntry> transactions = (List<HistoryEntry>) oi.readObject();
            entries.addAll(transactions);

            //delete file when done updating
            f.delete();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                fi.close();
                oi.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Figure 20: $fetchUpdate()$ method reads all updates sent by peers in files and append to the local copy of History Blockchain in $HistoryBC$ class