School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Implementation of a Memory Optimized and Flexible Blockchain

by

Aaron Chelvan

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Computer Engineering

Submitted:	November 2019	Student ID:	z5117237
Supervisor:	Salil Kanhere	Topic ID:	

Abstract

A Memory-Optimized and Flexible Blockchain (MOF-BC) is a blockchain variant described in the 2018 paper *MOF-BC: A Memory Optimized and Flexible BlockChain for Large Scale Networks* by Ali Dorri, Salil Kanhere, and Raja Jurdak. It allows for transactions that have been added to the blockchain to be removed at a later point in time, as a way of minimising the blockchain's memory footprint. This thesis project involved implementing a MOF-BC network and gathering results about its memory usage and performance, to determine its viability. MOF-BC was found to be viable under certain circumstances, and some techniques for further reducing the memory usage of the blockchain were explored.

Acknowledgements

I would like to thank Salil Kanhere and Ali Dorri for their supervision and guidance over the course of this thesis project.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) industry has been growing exponentially in recent years. The value of the global IoT market is expected to triple from 2016 to 2020, resulting in an estimated value of approximately $450 billion [Col17]. There are 3 main areas which account for more than two-thirds of the IoT global market share:

1. **Smart Cities** - Buildings may utilise energy management systems, which involve the usage of IoT sensors to monitor and control both the temperature and lighting within it. This can help minimise the amount of energy wasted. Some examples may include turning off the lighting in the areas of a building that have no detected occupants, or dynamically adjusting the temperature setting of the air conditioning system based on the temperature outside.

2. **Industrial IoT** - Industrial automation is becoming increasingly prevalent, and allows for increased productivity and reduced costs. IoT devices can be used to monitor the performance of manufacturing equipment, and offer more in-depth logistical information, among other uses.

3. **Connected Health** - IoT can change the way healthcare services are provided. For example, internet-connected asthma inhalers can record and transmit data about when and where a patient uses the inhaler. This data can be used by

the patient's physician to monitor their usage. Furthermore, data from multiple inhalers can be aggregated to extract common trends, and potentially guide future research into treatments and/or cures.

A blockchain is a distributed ledger that allows for data to be stored in such that a way that it cannot be modified. This data is stored in the form of transactions, with multiple transactions being grouped together into blocks. Each block is represented by an ID that is computed by hashing the contents of the block, which also includes the ID of the previous block added to the blockchain, hence creating a chain of blocks. This concept ensures that all data in the blockchain is immutable, since editing the contents of a block would alter the block ID, which would immediately be noticeable because it's original block ID is stored in the contents of the next block. This immutability makes the blockchain ideal for situations where the auditability of data is important, which is one of its main benefits. [CNP+15]

Blockchain networks are composed of nodes and miners. Due to the nature of a distributed ledger, there is no single authority that is responsible for storing and managing a blockchain. Instead, each of the miners will have a local copy of it and they will contribute to the network by providing their computing resources to mine new blocks to the blockchain. Nodes, on the other hand, simply create transactions which are sent to the miners for mining. This decentralized system prevents the issue of a central authority potentially being malicious.

However, typical blockchain implementations also possess some characteristics that make them unsuitable for use in an IoT context. Firstly, blockchains have relatively expensive memory requirements since all past transactions need to be stored such that the consistency of the chain of hashes can be verified. Figure 1.1 shows the exponential growth of the size of the Bitcoin blockchain over time.

IoT networks would typically consist of a large volume of transactions being transmitted, due to the sheer number of devices involved. This would only further amplify the rate at which the blockchain grows. At the same time, IoT devices generally have

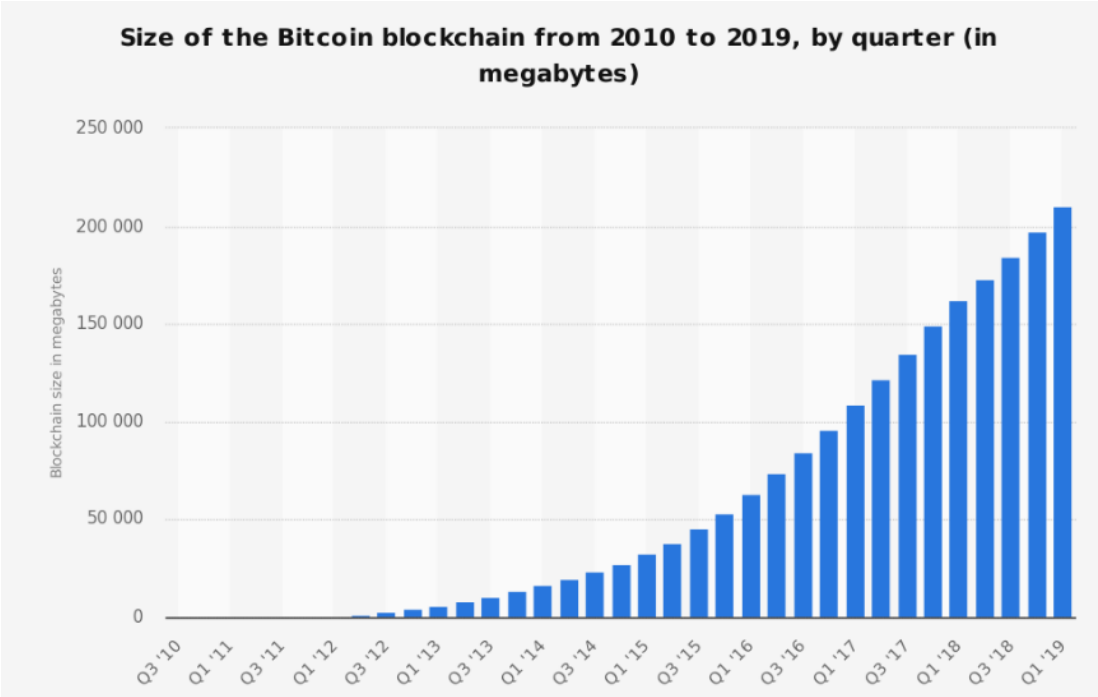

Figure 1.1: Growth of the Bitcoin Blockchain [Sta19]

constrained memory requirements which may make using a blockchain unfeasible.

Another issue is the relative lack of privacy associated with using a blockchain. Once a transaction is created and mined to the blockchain, that information is available to every node that has a copy of the blockchain and it cannot be removed or modified while preserving the chain of hashes. Although this allows for auditability, it comes at the cost of privacy, even if the identity of a user is kept anonymous. This is because every transaction contains the public key that corresponds to the user who created that transaction. Therefore, if a user employs the same public and private key pair for all transactions they create, then multiple transactions that contain the same public key can be linked to gather information about the identity of that user.

The Memory-Optimized and Flexible Blockchain (MOF-BC) is a blockchain variant which attempts to alleviate these issues. It allows for memory optimizations to be made to transactions after they have been mined to the blockchain, without invalidating the hash consistency. It also maintains the privacy of users by preventing transactions from containing any identifying information about the user who created that transaction.

The goal of this thesis is to evaluate the feasibility of MOF-BC as a blockchain variant for memory-constrained IoT devices.

Chapter 2

Background

This chapter will cover some of the concepts discussed in *MOF-BC: A Memory Optimized and Flexible BlockChain for Large Scale Networks* [DKJ18].

2.1 Overview

MOF-BC is a variant of typical blockchain implementations that allows for transactions to be removed, even after the transaction has been added to the blockchain. Removing transaction data serves two purposes: it can reduce the memory usage of the blockchain, and it can improve the level of privacy associated with using a blockchain (since the transaction no longer has to be permanently stored on the blockchain, and it can be removed once it is no longer needed). In order to provide this functionality, MOF-BC uses a variety of **agents** (programs which run on their own individual nodes within the blockchain network), with each agent carrying out a particular role. MOF-BC also introduces the concept of a **Generator Verifier**, which allows for users to remain anonymous when creating transactions and prevents multiple transactions from the same user to be linked together to gather information about that user's identity. This implementation of MOF-BC will support two types of memory optimizations:

- **Transaction removal** - when the data for a single transaction in the blockchain is removed.
- **Transaction summarization** - when the data for multiple transactions in the blockchain are removed all at once.

Users are only allowed to remove and summarize transactions that they themselves created.

2.2 Block Structure

Typical blockchain implementations do not allow transaction data to be modified, because doing so would break the hash consistency of the blocks. This is because the ID of a block is generated by hashing the contents of the entire block. Within the contents of the block will be the ID of the previous block, and this is how the blocks are chained together. However, MOF-BC allows for transaction data to be removed, and this is made possible due to the fact that the block ID is computed by only hashing the transaction IDs and previous transaction IDs of all transactions in the block. Figure 2.1 demonstrates the concept of how the removal of transaction data from a MOF-BC block will not change the block ID, and hence will not affect the hash consistency of the blockchain.

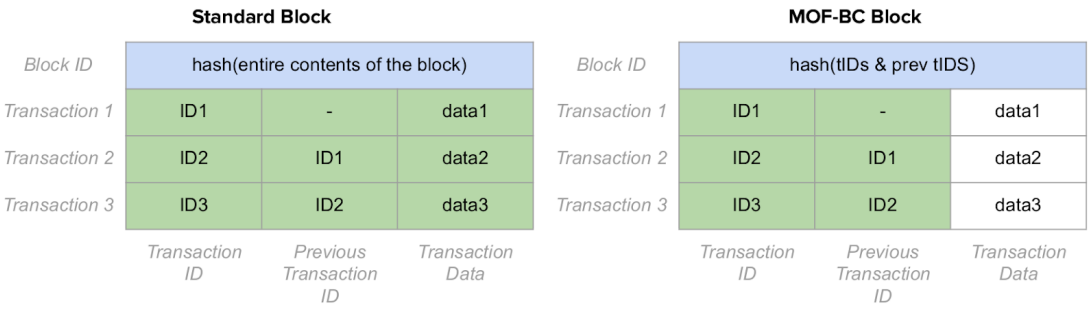

Figure 2.1: The computation of a block ID in a typical blockchain implementation and in MOF-BC

The removed data does not necessarily have to be deleted. The user could potentially

store that data somewhere off the blockchain (such as in a cloud storage service), and by hashing that data and comparing the hash with the transaction ID stored in the block (note that a transaction's ID is the hash of the data of that transaction), it can be proved that at some point in time, that data existed within the blockchain. The previous transaction IDs are also stored so that the order of the transactions in the block is maintained.

2.3 Generator Verifier

In standard blockchain implementations, all transactions will include a signature that was signed using the private key of the particular user that created that transaction. Thus, the legitimacy of the transaction can be verified by checking that the public key for that user (which is included in the transaction) can successfully decrypt the signature. However, this can lead to privacy concerns since if the same public and private key pair are used to sign multiple transactions, then it can be deduced that those transactions came from the same user and hence, it could potentially reveal information about that user.

In order to preserve privacy, one method would be for nodes to generate a new public and private key pair for every transaction they create. The issue with this is that the nodes would then have to store all of these key pairs because, if they later want to remove a transaction, they will have to prove that they created that transaction using the corresponding private key used to sign it. Due to the memory constraints associated with an IoT device, storing all of these keys is infeasible.

To solve this, every user has their own **Generator Verifier Secret** (GVS) and a **public and private key pair** (which will be referred to as GV-PK+ and GV-PK-, respectively). The GVS is some data that is known only to the user (i.e. a password). Every transaction in MOF-BC contains a **Generator Verifier** (GV), which is computed by appending the hash of the GVS to the hash of the previous transaction ID (P.T.ID), computing the hash of the result, and then signing that hash with GV-PK-

(i.e. the GV is computed as $sign(hash(hash(GVS)||hash(P.T.ID)))$). Since the P.T.ID will be unique for each transaction, this ensures that the GV will also be unique for each, and thus user privacy can be maintained.

MOF-BC only allows users to remove or summarize transactions that they themselves created. As a result, when a user attempts to perform one of those operations, the network's miners must verify that this is the same user who created the original transaction(s). All removal and summary transactions must contain the following fields:

1. **Unsigned GV** - An unsigned version of the GV that appears in the transaction being removed/summarized. The user computes this by hashing their GVS (which is a secret that they know), hashing the P.T.ID (the ID of the transaction previous to the transaction being removed/summarized), appending the hashes, and hashing the result.
2. **GV-PK+** - The public key that corresponds to the private key (GV-PK-) that was used to sign the GV in the transaction being removed/summarized.
3. **Signature** - Some data which has been signed using the same GV-PK- used to sign the GV in the transaction being removed/summarized.

Summary transactions will also require some additional fields, but that will be discussed in Section 2.6.

The verification process is as follows (where X is the removal/summary transaction, and Y is the transaction being removed/summarized):

```
if (X.GV-PK+ does not decrypt Y.GV) {
    return False
} else if (X.Unsigned-GV != decrypted Y.GV) {
    return False
} else if (X.GV-PK+ does not decrypt X.Signature) {
    return False
```

```
} else {
    return True
}
```

Verification involves attempting to decrypt the GV in transaction Y transaction with the public key in the transaction X. If that is successful, the decrypted GV is compared to the unsigned GV included in the transaction X. The third and final step checks that this user knows the private key used to sign the GV in transaction Y, hence proving that this user created transaction Y.

2.4 Agents

Agents are programs that run on their own individual nodes within the blockchain network. Each agent serves a role in carrying out the functionality of MOF-BC. This implementation of MOF-BC will include the use of three agents:

- **Search agent** - Searches the blockchain for removal transactions and summary transactions, and sends the locations of those transactions to the service agent or the summary manager agent respectively.
- **Service agent** - Receives locations of removal transactions from the search agent, and processes them as described in Section 2.5. Once processing is complete, any updates are transmitted to the miners.
- **Summary manager agent** - Receives locations of summary transactions from the search agent, and processes them as described in Section 2.6. Once processing is complete, any updates are transmitted to the miners.

Figure 2.2 shows how these agents work alongside a miner to process memory optimizations.

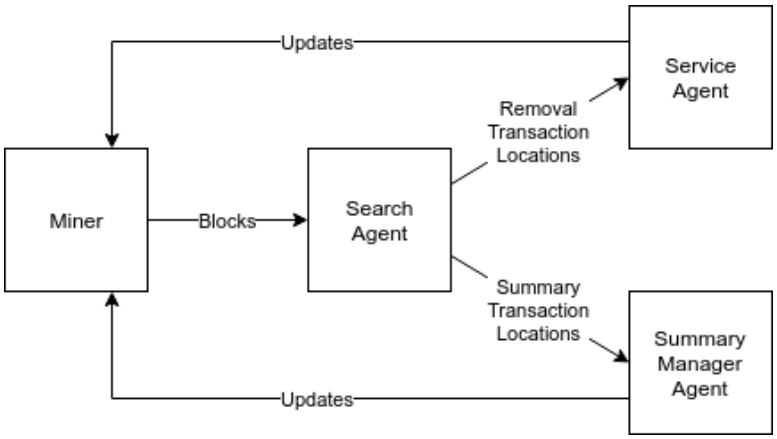

Figure 2.2: The flow of data between a miner and the agents

Within the MOF-BC network, there can be multiple instances of each agent. As the size of the network grows, more instances can be initialised and this ensures the scalability of the network.

2.5 Transaction Removal

One possible memory optimization that can be made is transaction removal. Firstly, the user will create a removal transaction which contains an unsigned GV, the GV-PK+, and a signature (as described in Section 2.3), as well as the ID of the transaction that is to be removed. Once it has been verified that the user who created the removal transaction is also the creator of the transaction being removed, a miner will add the removal transaction to the blockchain. Every time a new block is mined to the blockchain, that block will be transmitted to all of the agents in the network.

When the search agent receives a new block and finds a removal transaction within the block, it will notify the service agent of the ID of the transaction to be removed. The service agent will then scan through the entire blockchain, remove the data for that transaction, and broadcast this update to all of miners and other agents in the network so that they can all update their local copies of the blockchain.

2.6 Transaction Summarization

A summary transaction is similar to a removal transaction, except instead of having an individual removal transaction for each transaction being removed, a single summary transaction will contain information for multiple transactions being removed. To summarize a collection of transactions, the user must create a summary transaction which contains the following:

- A **timestamp** of when the summary transaction was created.
- A **signature**, the **GV-PK+**, and a list of **unsigned GVs** (one for each transaction being summarized) as described in Section 2.3.
- The **Merkle tree root** of the transactions being summarized. A Merkle tree is a data structure that is generated by computing the hashes of all of the transactions being summarized, appending those hashes pairwise, computing the hash of the results, and repeating that process until there is a single hash remaining (which is referred to as the tree root). This final hash is what is included in the summary transaction. Figure 2.3 demonstrates the process of computing the tree root.

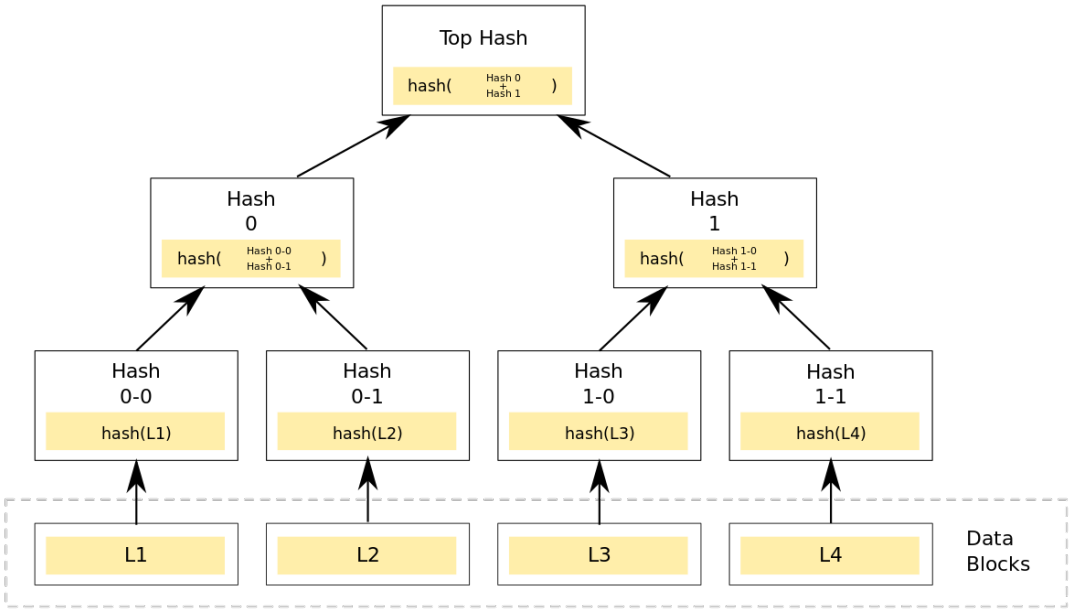

Figure 2.3: The computation of a Merkle tree root [Wik19]

- **Summary-time** - A list of timestamps of when each of the transactions being summarized were created. These timestamps are stored in chronological order.
- **TransOrder** - A list used to map each transaction being summarized to a timestamp in the Summary-time list. The Merkle tree can be used to prove that a given transaction exists within the group of transactions being summarized, but it does not provide information about the chronological order of when those transactions were created. One such method to maintain the order could be to store an ordered list of the transaction IDs of the transactions being summarized, so that the first transaction in the list maps to the first timestamp in the Summary-time list, and so on. However, storing the entire transaction ID is not necessary. Instead, storing the minimum possible number of distinct bytes in the transaction IDs is sufficient. For example, Figure 2.4 shows 4 transactions and their corresponding transaction IDs. Transactions 1 and 3 share the same first 2 bytes in their IDs, but the third bytes are different. Hence, only the first 3 bytes of each transaction ID needs to be stored to preserve the order of the transactions.

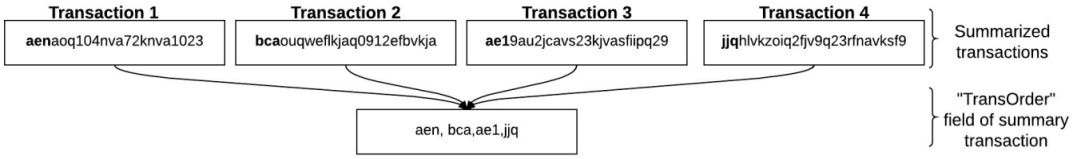

Figure 2.4: The computation of the TransOrder list [DKJ18]

To summarize a set of transactions, a user will create a summary transaction containing all of the fields described above. A miner will then verify each transaction being included in the summary to check that this user is the creator of that transaction. Once the summary transaction is added to the blockchain, the search agent will notify the summary manager agent, which will traverse through the entire blockchain, and remove the transaction data of all of the transactions that were summarized. These updates are transmitted to the miners and other agents, who then then update their local copies of the blockchain to reflect those changes.

2.7 Cleaning Period

Every time a removal transaction or summary transaction is created, the service agent or summary manager agent must search through their local copies of the blockchain and remove the corresponding transaction data. If the agents were to do this as soon as they receive a removal/summary transaction, the processing associated with it would be costly because the agents need to scan the entire blockchain (which occurs in linear time) to find the transaction they are looking for. To minimise this, MOF-BC networks employ a **Cleaning Period** (CP) which is an interval of time that is specified when the network is initially configured. At the end of each CP interval, the service agent and summary manager agent perform a single scan of their copy of the blockchain and remove all of the corresponding transaction data for all of the memory optimizations that have accumulated over the past CP interval.

The value that the CP should be set to will vary depending on the data that the MOF-BC network is being used to store. Larger CPs result in the blockchain growing to

a larger size in between CP intervals, since more removal transactions and summary transactions can accumulate. But in the long term, it will not have any impact on the size of the blockchain.

Chapter 3

Implementation

This chapter will cover the design of the MOF-BC network that was implemented, and provide a description of the technologies used.

3.1 System Design

The MOF-BC network was implemented by writing separate programs for the node, miner, and each of the agents, each of which run within their own containers. These containers were run on the same host machine. Figure 3.1 describes the overall design of the network.

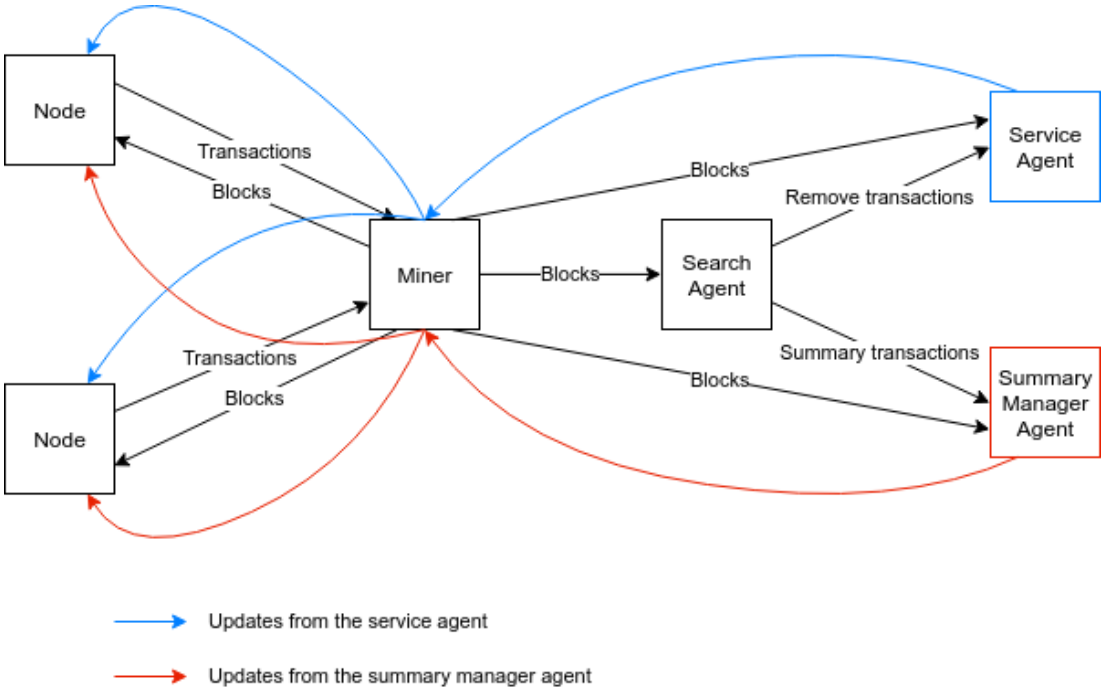

Figure 3.1: The system design of a MOF-BC network

The arrows represent the flow of data between the containers. This implementation supports one miner, one instance of each of the agents, and an arbitrary number of nodes (however, Figure 3.1 shows an example network of only two nodes).

The nodes transmit transactions to the miner. The miner adds those transactions to a block, and when a block is filled, it is broadcast to all nodes and agents in the network. The search agent will scan the newly mined blocks for remove transactions and summary transactions, and transmit the locations of those transactions to the service agent and summary manager agent respectively. The service agent will process the remove transactions, and transmit the updated blocks to the miner, which will then relay those updates to the nodes. The summary manager agent will do the same for the summary transactions.

3.2 Communication

All communication between the nodes, miner, and agents is achieved over TCP socket connections. They all need to be able to accept incoming connections (i.e. behave as a TCP server) while simultaneously initiating connections to others in the network (i.e. behave as a client). As a result, all of the programs need to be multithreaded to allow for both functions to occur concurrently.

For example, the node program needs to create transactions and transmit them to the miner, while also receiving mined blocks and updated blocks (blocks which contain transactions that have been removed or summarized) from the miner. The server component of the program consists of a loop which continually accepts incoming socket connections from the miner. In a separate thread, the function that creates and sends transactions is executed at regular intervals.

3.3 Software Design

The programs for the node, miner, and agents, were implemented using Java 8. Java was chosen for a number of reasons:

- **Compiled language** - Java is a compiled language and hence, it will typically be more performant than a scripting language. This is helpful because during the benchmarking stage of this project, performing tasks such as filling a blockchain with millions of transactions will be less time consuming. Also, compiled languages are generally safer since errors will be identified at compile-time instead of run-time, which helps speed up the development process.
- **Object-oriented** - Java is an object-oriented language and the implementation of a blockchain network is suited for an object-oriented design. This implementation of MOF-BC was designed by creating the following classes: Transaction, Block, Node, Miner, SearchAgent, ServiceAgent, and SummaryManagerAgent.

- **Object serialization** - Java allows for objects to be easily serialized (converted into an array of bytes) and deserialized (converted from a byte array back into an object). Serialization is necessary for transmitting objects over a network or writing them to a file. To serialize an object, that class needs to implement Java's **Serializable** interface. The Transaction class implements this interface so that when a node wants to transmit a transaction to the miner, the node can create a Transaction object, serialize the object, and transmit it over a socket connection to the miner. When the miner receives the serialized transaction, it will deserialize it before adding it to the current block. The Block class also implements the Serializable interface so that when a miner has filled a block with transactions, it can broadcast the newly-mined block to the entire network, as well as write the block to its local copy of the blockchain (this will be discussed in further detail in Section 3.4).

- **Security** - Java's **java.security** package contains a number of security-related functions that are necessary for implementing some of MOF-BC's functionality. Specifically, functions for generating an RSA key pair, as well as encrypting, decrypting, signing, and verifying data using this key pair.

Figure 3.2 shows the structure of the project directory (some files have been omitted for clarity). Within the main project directory, there are separate subdirectories for the node program, the miner program, and each of the agents. Within each of these subdirectories is the source code for that particular program as well as a Dockerfile for it (Dockerfiles are explained in Section 3.5.1). There is also a subdirectory named *common* which contains source code that is shared between all of the programs (e.g. the Transaction class and the Block class). When each of the programs are compiled, the files in the *common* subdirectory will be compiled with the files in the subdirectory specific to that program.

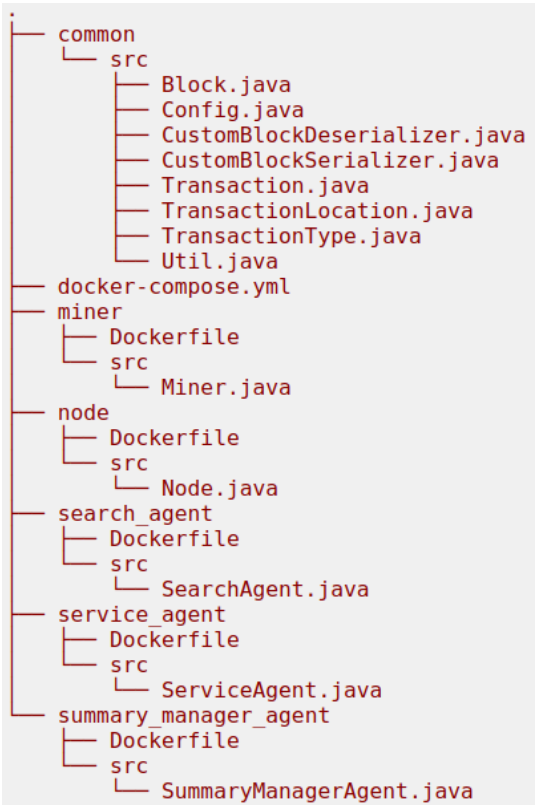

Figure 3.2: The project directory structure

The *docker-compose.yml* file is explained in Section 3.5.2.

3.4 Blockchain Storage

The nodes, miner, and agents all store their own local copy of the blockchain. **LevelDB** is a high-performance key-value storage library which is used for blockchain storage within this MOF-BC implementation [GD19]. It was chosen because it provides a very simple interface for writing to and reading from the database. All data is stored in the form of key-value pairs, and database manipulation occurs via the following API functions:

- **put(key, value)** - stores a key-value pair in the database
- **get(key)** - retrieves the value corresponding to the given key

- **delete(key)** - deletes the value corresponding to the given key
- **iterator()** - iterates through all of the keys in the database

Blocks are stored in the database by using the block ID as the key, and the serialized block object as the value. Also, since LevelDB is a C++ library, **LevelDB JNI** (an open-source Java interface) [Fus19] was needed for these programs to be able to use a LevelDB database.

3.5 Containers

3.5.1 Docker

A container is a package of software that consists of the source code for a program, the relevant external libraries, the environment variables, and any other necessary data required for the program to run. This greatly simplifies the process of deploying software since the machines running the program do not need to have all of the dependencies installed beforehand [Doc19b]. Docker is a tool for building and running containers. When Docker containers are run on a host machine, they operate similar to virtual machines, but the difference is that the containers share the same kernel and the same system libraries, and this significantly reduces the computational resources needed to run several containers simultaneously [Ope19].

The nodes, the miner, and the agents, all run within their own Docker containers on the same host machine. The alternative to using this containerized approach would have been to have the programs running on a group of physical devices (e.g. Raspberry Pis), but the issue with this is that every time a change is made to the software, the updated code needs to be loaded onto all of the devices and this can be time consuming. Furthermore, all of the devices need to be monitored in the event that one of the programs throws an error and exits. One such method of doing this could be to have multiple terminals open on a single machine, with each terminal having an

active SSH connection to each of the devices. However, managing of all these separate terminals can be inconvenient. By using containers, the issues with deploying the MOF-BC network to a set of physical devices can be avoided, speeding up the development process.

Before a Docker container can be run, a Docker image for that container needs to be built first. This is done by writing a Dockerfile which contains a set of instructions describing how the image should be built. Figure 3.3 shows the Dockerfile for the image of a node.

```
FROM openjdk:8
COPY node/src/ /usr/src/myapp
COPY common/src/ /usr/src/myapp
WORKDIR /usr/src/myapp
RUN javac -cp ".:./leveldbjni-all-1.8.jar" Node.java Block.java Transaction.java
CMD ["java", "-cp", ".:./leveldbjni-all-1.8.jar", "Node"]
```

Figure 3.3: The Dockerfile for the Node program

The first line specifies the base image which this Docker image should be built on top of. In this case, the base image has Java 8 installed. The second line indicates that all files in the *node/src/* directory (which contains source files specific to the node program) in the current working directory should be copied to the */usr/src/myapp* directory in the container image. The third line is similar to the second, except it instructs the files in the *common/src/* directory (which contains source files which are shared between the node, miner, and agent programs) to be copied over. The fourth line changes the working directory to the directory in the container image where all of the source files have been copied to. Any following instructions in the Dockerfile will now be executed in that directory. The fifth line will run a command that compiles the program. The arguments to *javac* include a number of Java files and a Java archive file for the LevelDB Java interface. The final line runs the node program which was just compiled on the previous line.

3.5.2 Docker Compose

Docker Compose is a tool for running multi-container applications. Running a group of containers using Docker Compose is achieved by creating a YAML file named *docker-compose.yml* which contains the necessary configuration data, running the command `docker-compose build` to build all of the Docker images, and running the command `docker-compose up` to run the Docker images. By default, when the containers are run, Docker Compose will create a single network for those containers to run on, and this allows the containers to communicate with each other [Doc19a]. This networking capability is important for achieving the necessary functionality of this project.

Figure 3.4 shows an example docker-compose.yml file. In this example, there is one instance of a miner and two instances of nodes.

```
version: '3'
services:
  node1:
    environment:
      PUB_KEY: "${NODE1_PUB_KEY}"
      PRIV_KEY: "${NODE1_PRIV_KEY}"
      GVS: "${NODE1_GVS}"
    build:
      context: .
      dockerfile: node/Dockerfile
  node2:
    environment:
      PUB_KEY: "${NODE2_PUB_KEY}"
      PRIV_KEY: "${NODE2_PRIV_KEY}"
      GVS: "${NODE2_GVS}"
    build:
      context: .
      dockerfile: node/Dockerfile
  miner:
    build:
      context: .
      dockerfile: miner/Dockerfile
```

Figure 3.4: An example Docker Compose YAML file

The *version* field contains the version of Docker Compose being used. Within the *services* field is a list of services which will be run together, with each service corresponding to a container. All of the services have a *build* field which specifies the location of the Dockerfile for that container. The two node services have some environment variables passed in (namely, a public key, a private key, and a Generator Verifier Secret). These

environment variables are saved in a file named *.env* in the same directory as this YAML file.

Chapter 4

Results

A number of tests were performed to measure the memory usage and performance of this implementation of MOF-BC. This section will discuss those results. For all of the tests, the MOF-BC network was run on a single laptop with an Intel Core i5-7200 CPU (2.50GHz) and 8GB of RAM.

4.1 Size of a Removal Transaction

One aspect of this MOF-BC implementation that was measured was the size of a removal transaction. It was found to be approximately 1.6KB. This means that if the size of the transaction being removed is less than 1.6KB, then the amount of memory saved by removing this transaction is outweighed by the memory occupied by the removal transaction. It is only when the size of the transaction being removed is greater than 1.6KB that memory savings can be obtained. The reason why a remove transaction is this large is because it contains the following pieces of data:

- *Unsigned GV (SHA-256 hash)* - 32 bytes
- *GV-PK+ (2048 bit public key)* - 256 bytes
- *Signature created with GV-PK-* - 256 bytes

These give a total of 544 bytes. The remaining size comes from Java's serialization algorithm [Jav09]. When an object is serialized, the resulting byte array includes:

- The class name
- A list of all attributes in the class. This includes:
  - The name
  - The type
  - The value
- Other metadata

The size of a serialized object can get quite large, especially if the type of one of the attributes is more complicated than a primitive data type. For example, if an attribute is a hashmap, then the serialized object will also need to contain the key type and value type of the hashmap, as well as all of the key-value pairs stored.

Given that MOF-BC was originally designed with IoT devices in mind, it would be ideal if the size of a remove transaction could be reduced to lower the minimum transaction size needed for a transaction removal to be worthwhile. Removal transactions contain an unsigned GV, a public key, and a signature, and none of these can be taken out because they are all required for verifying that the node that created the removal transaction is the same node that created the transaction to be removed. So, the only way the size of a removal transaction can be reduced is by using a different object serialization method. Section 4.5 discusses the results obtained from using some alternative serialization algorithms.

4.2 Size of a Summary Transaction

The size of a summary transaction was measured by varying the number of transactions being summarized into a single summary transaction. Figure 4.1 shows the correlation

between the number of transactions being summarized and the size of the summary transaction.

Number of transactions being summarized	Size of summary transaction (KB)	Size of summary transaction / number of transactions (KB)
1	2.0	2.0
5	2.8	0.56
20	5.7	0.29
100	21.6	0.22
200	41.7	0.21

Figure 4.1: The size of a summary transaction

It can be seen that as the number of transactions being summarized increases, the size of that summary transaction will also increase. This is to be expected since summary transactions contain both the timestamp and unsigned GV for each transaction in the summary, so the size of the summary transaction has an approximately linear relationship with the number of transactions in the summary. This linear relationship does not exist for summary transactions which summarize a small number of transactions (e.g. a summary transaction of 5 transactions is only 40% larger than a summary transaction of 1 transaction, despite the fact that it summarizes 5 times as many transactions). This is due to the size of other fields in the summary transaction (the timestamp, the Merkle tree root, etc.). However, as the number of transactions grows, the relative size of these other fields become negligible in comparison to the growing list of timestamps and unsigned GVs, which is why the linear relationship becomes clear for summary transactions that summarize 100 or more transactions.

When considering the ratio between the size of a summary transaction and the number of transactions it summarizes, it can be seen that the size per transaction falls as the number of transactions increases, up until a certain threshold. After about 100 transactions (i.e. once the linear relationship comes into effect), the size per transac-

tion plateaus at about 0.2KB. This means that one should try to summarize as many transactions as possible into a single summary transaction, in order to maximise the memory savings attained.

4.3 Performance of Transaction Removal

The performance of transaction removal was measured by instantiating blockchains consisting of 100, 500, 1000, and 2000 transactions and measuring the time taken to remove all of the transactions in them. These blockchains contained 100 transactions per block and each transaction contained 100 bytes of data, excluding transaction metadata. Also, the transactions were removed in a random order (as opposed to performing a single linear scan of the blockchain and removing each transaction in order) to provide results that are representative of a realistic use case of a MOF-BC network. Figure 4.2 displays these results. It can be seen that the time taken to remove a single transaction was consistently about 6-7 milliseconds, and there is a linear relationship between the number of transactions being removed and the time taken to remove them.

Number of transactions in the blockchain	Time taken to removal all transactions (ms)	Time taken per transaction (ms)
100	666	6.66
500	3597	7.19
1000	5930	5.93
2000	11319	5.66

Figure 4.2: The time taken to remove all transactions in a block

Another aspect of transaction removal that was examined was whether or not the number of blocks in a blockchain has an impact on the performance of transaction removal. This was measured by instantiating a number of blockchains with 2000 transactions

each, but varying the number of blocks that those transactions are split between. Figure 4.3 shows the time taken to remove all of the transactions in these blockchains.

Number of blocks	Number of transactions per block	Time to remove all transactions (seconds)
1	2000	7.77
2	1000	10.5
20	100	11.3
200	10	25.3
400	5	37.6

Figure 4.3: The time taken to remove 2000 transactions from blocks of varying sizes

As the number of blocks increases, the time taken to remove all of the transactions increases as well, despite the fact that the same number of transactions are removed. This result is to be expected since this implementation of MOF-BC stores the blockchain in a database, and every time a transaction needs to be removed from the blockchain, the following needs to occur:

- The entire blockchain needs to be scanned linearly until the block holding the transaction to be removed is found.
- That block needs to be read from the database and deserialized into a block object.
- The block object has to be modified to remove the relevant transaction.
- The newly modified block is written back to the database.

More blocks leads to more reading from the database file and more writing to the database file, and this additional overhead is the reason for the longer removal times for blockchains with more blocks. In order to minimise this overhead, one should aim to maximise the number of transactions per block as much as feasibly possible.

4.4 Performance of Transaction Summarization

The performance of transaction summarization was also measured, and this was done by instantiating a blockchain of 2000 transactions (20 blocks, with 100 transactions per block) with each transaction containing 100 bytes of data (excluding transaction metadata). A single summary transaction was created by having it summarize a random selection of transactions in the blockchain. This test was repeated by varying the number of transactions being summarized into a single summary transaction. Figure 4.4 shows the time taken to create a summary of varying numbers of transactions. It can be seen that there is a clear linear relationship between the number of transactions being summarized and the time taken to create that summary transaction (approximately 20 milliseconds per transaction).

Number of transactions per summary transaction	Time taken to create summary transaction (ms)	Time taken to create summary transaction / number of transactions (ms)
5	115.9	23.2
10	222.4	22.2
100	2099.1	21.0
500	10115.0	20.2
1000	21571.7	21.6
2000	39908.0	20.0

Figure 4.4: The time taken create summary transactions summarizing varying numbers of transactions

This linear relationship arises because creating a summary transaction involves scanning the blockchain for all of the transactions which are to be summarized within it. As a result, more transactions means more time has to be spent searching through the blockchain, and from Figure 4.4, it can be seen that the average time to find a particular transaction within the blockchain (for a blockchain of this size) is roughly

20 milliseconds. Combining these results with those in Section 4.2 shows that although maximising the number of transactions in a summary will maximise the memory savings achieved (per transaction), the tradeoff is that it will require more processing time.

4.5 Block Serialization

In Sections 4.1 - 4.4, the MOF-BC implementation used Java's object serialization interface to serialize blocks and store them in the blockchain. Although this interface makes it very easy for developers to serialize and deserialize any Java object, it was not designed to be as memory efficient as possible. A serialized object contains some unnecessary information that does not need to be included in the blockchain (such as the name of the class that the object belongs to, and the data type of each of the fields within the object). To avoid this additional information, some alternative serialization algorithms were explored: JSON (JavaScript Object Notation), CSV (comma-separated values), and a custom-made serialization algorithm. For each of these 3 algorithms, two functions were written: one function for serializing a Java block object into the corresponding serialization format, and another function for deserializing it back into a Java block object. The former function would get called whenever a node is about to store a block in its database, and the latter function is called when a node reads a block from its database. No other changes were made to the MOF-BC network's implementation.

4.5.1 JSON

The memory usage of a JSON-encoded block was measured by creating a block and serializing it using Java's serialization algorithm, creating another block and serializing it using JSON, and comparing the sizes of the two serialized blocks. This was repeated for blocks containing various numbers of transactions. Each transaction contained 100 bytes of data, excluding transaction metadata. Figure 4.5 shows how the blocks have been formatted using JSON. The main object contains values for the block ID, previous

block ID, and a list of transaction objects. Each transaction object contains all of the values for the corresponding transaction.

```
{
  "blockId": ...,
  "prevBlockId": ...,
  "transactions": [
    {
      "tid": ...,
      "prevTid": ...,
      ...
    },
    {
      "tid": ...,
      "prevTid": ...,
      ...
    },
    ...
  ]
}
```

Figure 4.5: The format of a block when encoded into JSON

Number of transactions in block	Size of block (Java serialization) (bytes)	Size of block (JSON) (bytes)
10	5770	6756
100	53020	66336
1000	525520	662136

Figure 4.6: Block sizes when serialized using Java's serialization algorithm and when serialized using JSON

Regardless of the number of transactions in a block, it can be seen in Figure 4.6 that the size of the JSON-encoded block is approximately 20% larger than the equivalent Java-serialized block. The main reason for this is that JSON does not allow raw binary data to be stored within it, because quotation marks, commas, brackets, and other symbols are used for formatting the data in a JSON file, and raw binary data could potentially contain one of these special characters. One possible way to store binary data is to encode it using Base64 (so that all binary data can be encoded using a character set consisting only of alphanumeric characters, plus symbols (+) and forward slashes (/)), which is what was done for the purpose of this test. The problem with

Base64 is that it is a relatively small character set which means that more bytes may be needed to represent some binary data in Base64 compared to representing that same data as raw binary. For example, `f32abd` (which is the hexadecimal representation of 3 arbitrary bytes) has a Base64 encoding of `8yq9`. In other words, for this particular example, the data which was only 3 bytes long in raw binary format is now 4 bytes long when encoded to Base64. This explains why the JSON-encoded blocks are considerably larger than the Java-serialized ones.

4.5.2 CSV

CSV was also explored as a potential serialization algorithm for encoding blocks. The memory usage of a CSV-encoded block was measured using a similar procedure to that described in Section 4.5.1: create a block and serialize it using Java's serialization algorithm, create another block and serialize it using CSV, compare the sizes of the two serialized blocks, and repeat these steps for blocks containing varying numbers of transactions. Each transaction contains 100 bytes of data, excluding transaction metadata. Figure 4.7 shows how the blocks have been serialized in CSV. The first line contains the block ID and previous block ID. Each of the transactions in the block are written to a separate line, with each line containing all of the information corresponding to that transaction. The ordering of the values within a line (the transaction ID is the first value, and the previous transaction ID is the second value, etc) is defined in the serialization and deserialization functions. This means that only the values themselves need to be stored in the CSV-encoded block, because when it is deserialized, the deserialization function will know what each value corresponds to.

```
<blockId>,<prevBlockId>
<tid>,<prevTid>,...
<tid>,<prevTid>,...
...
```

Figure 4.7: The format of a block when encoded into CSV

Number of transactions in block	Size of block (default serialization) (bytes)	Size of block (CSV) (bytes)
10	5770	6100
100	53020	60190
1000	525520	601090

Figure 4.8: Block sizes when serialized using Java's serialization algorithm and when serialized using CSV

As with JSON, CSV files cannot store raw binary data, and so the contents of the blocks were encoded in Base64 before being written to a CSV string. Figure 4.8 shows the results obtained, and it can be seen that CSV-encoded blocks are larger than their Java-serialized equivalents, and this is due to the same reason described in Section 4.5.1, where Base64 encoding causes the length of binary data to increase. That being said, the size of a CSV-encoded block is smaller than the size of a JSON-encoded block with an equivalent number of transactions (from the results in Figure 4.6). This arises due to the fact that CSV is simply a list of values separated by commas and newlines, whereas JSON uses key names, brackets, quotes, and other characters to format the structure of the data.

4.5.3 Custom Serialization

Neither JSON encoding nor CSV encoding were able to successfully reduce the amount of memory a block occupies. In both cases, the fact that binary data needed to be encoded into Base64 before it could be stored in a JSON/CSV file is what caused the JSON-encoded and CSV-encoded blocks to be larger than the Java-serialized equivalents. To develop a serialization algorithm that is more efficient than Java's serialization algorithm, a custom serialization algorithm that does not require binary data to be encoded into Base64 needed to be designed. As a starting point, Java's serialization algorithm was examined.

Figure 4.9 shows an example Java class named *TestSerial* that implements the Serializable interface (described in Section 3.3) along with the binary data (in hexadecimal format) produced when an instance of this class is serialized [Jav09].

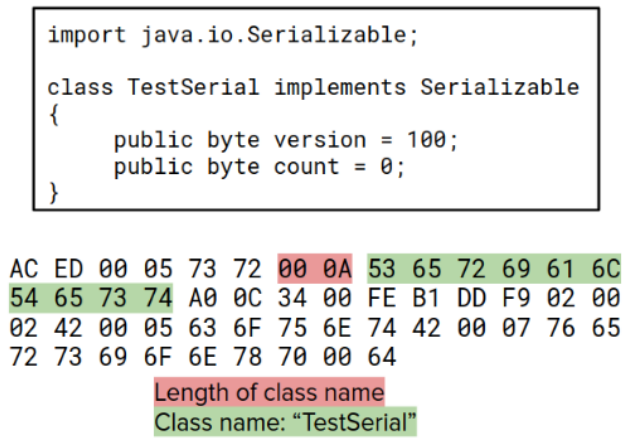

Figure 4.9: An example Java class and the corresponding serialized data

From Figure 4.9, `000A` (the 7th and 8th bytes) indicates the following 10 bytes (`000A` is 10 in hexadecimal) represent the name of this class. Those 10 bytes (`53657269616c54657374`) are the hexadecimal ASCII codes for the name *SerialTest*. The remaining information has been encoded in a similar format, where the number of bytes that a particular value occupies directly precedes the value itself so that when deserialization occurs, the deserializer knows how many bytes it needs to read to finish reading a particular value.

The custom serialization algorithm uses this technique to store raw binary data and separate the values without having to encode the data in Base64. The format of a serialized block using this algorithm is almost identical to the CSV format in Section 4.5.2, but there are two main differences:

1. Instead of separating values by commas and newlines, each value is directly preceded by 4 bytes representing an unsigned integer which contains the number of bytes that make up the following value.

2. All data is stored in raw binary format (without encoding it to Base64).

To measure the memory usage of this custom serialization algorithm, 3 blocks with varying numbers of transactions were created. Each transaction contained 100 bytes of data, excluding metadata. Figure 4.10 shows the size of these blocks when they are serialized using Java's serialization algorithm as well as the custom serialization algorithm. Unlike with JSON and CSV, the custom serialization algorithm was able to reduce the size of a block to be smaller than what Java's serialization algorithm could achieve. These memory savings can be attributed to the fact that the custom algorithm avoids storing any Java-specific information in the serialized block (e.g. the class name, and the data types of each of the values in the object), and instead, it only stores the bare minimum amount of information needed to be able to recreate the original block object, while at the same time not having to encode binary data into Base64.

Number of transactions in block	Size of block (default serialization) (bytes)	Size of block (custom serialization) (bytes)
10	5770	4672
100	53020	46072
1000	525520	460072

Figure 4.10: Block sizes when serialized using Java's serialization algorithm and when serialized using the custom algorithm

4.5.4 Performance

A factor to take into account when using either the JSON, CSV, or custom serialization algorithms is that every time a block is written to the database, it needs to be serialized appropriately and when a block is read from the database, it needs to be deserialized appropriately. Both serialization and deserialization takes some time to complete processing. Figure 4.11 displays the average time taken to serialize a block object using JSON, CSV, and the custom algorithm, as well as the average time to deserialize back to the block object. Each transaction contained 100 bytes of data, excluding transaction metadata. The measurements were taken for blocks with varying

numbers of transactions.

Number of transactions in block	10	100	500	1000
Time to serialize to JSON (ms)	0.85	1.43	1.92	4.90
Time to deserialize from JSON (ms)	1.53	3.62	4.87	9.40
Time to serialize to CSV (ms)	0.35	1.35	6.79	5.31
Time to deserialize from CSV (ms)	0.44	3.41	6.81	7.53
Time to serialize to custom serialization (ms)	0.25	1.01	1.98	3.85
Time to deserialize from custom serialization (ms)	0.33	0.92	2.03	3.00

Figure 4.11: Time taken to serialize and deserialize blocks of varying sizes

Some notably observations from Figure 4.11 are:

- For JSON and CSV, deserialization generally takes longer than serialization. The same conclusion cannot be made about the custom algorithm.
- For all three serialization types, as the number of transactions in a block increases, the time to serialize and deserialize it also increases.
- The time to serialize and deserialize a block to and from the custom serialization format is noticeably shorter than both JSON and CSV, especially for blocks with larger amounts of transactions.

Although these serialization and deserialization times are only a few milliseconds each, they can add up quite quickly after the MOF-BC network has been running for a long period of time.

Chapter 5

Conclusion

The goal of this project was to determine whether or not MOF-BC is a feasible blockchain variant that can be applied in situations involving memory-constrained devices (such as IoT devices). For this particular implementation, it can be concluded that MOF-BC is feasible under certain circumstances.

Firstly, the size of a removal transaction is 1.6KB which means that if one wishes to remove a transaction that is smaller than 1.6KB, then the memory savings obtained from removing it would be outweighed by the memory occupied by the removal transaction. As a result, this MOF-BC implementation is more suitable for situations where the transactions being added to the blockchain are relatively large (to be specific, larger than 1.6KB). If a transaction is smaller than 1.6KB, then removing it becomes unviable.

The size of a summary transaction increases as the number of transactions being summarized within it increases. However, the size divided by the number of transactions falls as the number of transactions in a summary increases. Once the number of transactions reaches about 100, the size per transaction plateaus at about 0.2KB per transaction. In other words, if a transaction is smaller than 0.2KB, then summarizing it will never be viable. This means MOF-BC is unsuitable for blockchain networks where transactions will regularly be smaller than 0.2KB. Also, one should aim to summarize as many transactions together as possible in order to maximise memory savings. Fur-

thermore, if there is more than one transaction to be removed, it will always be more memory efficient to summarize them together as opposed to creating separate removal transactions for each of them.

Although MOF-BC provides the opportunity to reduce the memory footprint of the blockchain, there are some performance considerations that need to be taken into account. Transaction removal takes approximately 6ms to complete and transaction summarization takes about 20ms per transaction being included in the summary (note that these timings were obtained for a blockchain with 100 transactions per block and 100 bytes of data per transaction, excluding metadata). These are relatively short times, but over a long period of time, they can accumulate and so, it must be understood that although MOF-BC can provide memory savings, there is a slight performance tradeoff.

An optimization that can be made to a MOF-BC network to improve its performance is to maximise the number of transactions that a single block can contain (within reasonable limits). The reason for this is that if there are two blockchains with 2000 transactions, but one of them as 10 blocks with 200 transactions each, and the other has 200 blocks with 10 transactions each, then the former blockchain will be more performant when it comes to actions such as scanning through the entire blockchain to find and remove a particular transaction. This is because fewer blocks means fewer reads from the database and fewer writes to the database, but the downside of fewer blocks with more transactions per block is that memory-constrained IoT devices may not have the RAM to load a large block into memory. As a result, the maximum number of transactions that can be feasibly included in a block is dependent on the type of hardware being used as nodes for a MOF-BC network.

In this implementation, the blocks were stored in a database by serializing them using Java's serialization algorithm. Although this algorithm allows developers to easily serialize objects in a majority of situations, it was not designed to be as compact as possible. A custom serialization algorithm was designed and it proved to be more memory efficient than Java's serialization algorithm. The tradeoff is that serializing and deserializing a block using this custom algorithm takes a few milliseconds each

time. If one wishes to minimise the memory footprint of the blockchain, they may decide to use this custom algorithm, but it will come at the cost of processing time.

Overall, MOF-BC is a viable blockchain variant for memory-constrained devices if the transactions being stored are large enough such that performing either a transaction removal or transaction summarization is worthwhile. Despite this, performing either one of these memory optimizations will require additional computational resources and time, and whether or not this is worth the memory savings that can be gained depends on the situation in which MOF-BC is used.

Bibliography

[CNP+15] Michael Crosby, Nachiappan, Pradhan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. *Blockchain Technology - Beyond Bitcoin*. Sutardja Center for Entrepreneurship Technology, Berkeley Engineering, 2015.

[Col17] Louis Columbus. 2017 roundup of internet of things forecasts. www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts, accessed 11/4/2019, 2017. Forbes.

[DKJ18] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. *MOF-BC: A Memory Optimized and Flexible BlockChain for Large Scale Networks*. School of Comp. Sci. and Eng., UNSW, 2018.

[Doc19a] Docker. Overview of docker compose. https://docs.docker.com/compose/, accessed 4/8/2019, 2019.

[Doc19b] Docker. What is a container? https://www.docker.com/resources/what-container, accessed 2/8/2019, 2019.

[Fus19] FuseSource. Leveldb jni. https://github.com/fusesource/leveldbjni, accessed 29/7/2019, 2019. GitHub.

[GD19] Sanjay Ghemawat and Jeff Dean. Leveldb. https://github.com/google/leveldb, accessed 29/7/2019, 2019. GitHub.

[Jav09] JavaWorld. The java serialization algorithm revealed. https://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html, accessed 4/8/2019, 2009.

[Ope19] Opensource.com. What is docker? https://opensource.com/resources/what-docker, accessed 2/8/2019, 2019.

[Sta19] Statista. Size of the bitcoin blockchain from 2010 to 2019, by quarter (in megabytes). https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/, accessed 12/4/2019, 2019.

[Wik19] Wikipedia. Merkle tree. https://en.wikipedia.org/wiki/Merkle_tree, accessed 14/4/2019, 2019.