

Developing A Blockchain-based System For Storing Data From The IoV-like Network Emulation

Student Name: Yile Feng

Candidate Number: 246743

Supervisor: Dr. Naercio Magaia

Degree: BSc Computer Science and Artificial Intelligence

Department: Informatics

April 2023

Statement of Originality

This report is submitted as part of the requirements for the degree in Computer Science and Artificial Intelligence at the University of Sussex. It is the product of my own labour, except as noted in the text. This report may be freely copied and distributed, but credit should be given to the source. I hereby give permission for copies of this report to be loaned to students in future years.

Signed

Abstract

The aim of this project is to develop a model blockchain system capable of storing the results of IoV-like network simulations. This system is intended to provide a prototype for blockchain systems for the preservation of IoV data. The system is written in python and contains basic blockchain features, including creating accounts, load blockchain, send transactions, P2P and so on. In addition to this, there are also features about automatically receiving and encrypting data on the chain that are added together. The user interface has been designed so that users can access the blockchain system normally through it.

1. Contents

Statement of Originality	2
Abstract	3
1. Contents	4
2. Introduction	5
2.1 Motivation	5
2.2 Problems	5
2.3 Obejectives	5
3. Professional Considerations and Ethics	5
3.1 Professional consideration	6
3.2 Ethical considerations	6
4. Background information	7
4.1 Blockchain	7
4.1.1 <i>Why we need blockchain?</i>	7
4.1.2 <i>How does blockchain works?</i>	8
4.2 Consensus algorithm	9
4.2.1 <i>Proof of Work (PoW)</i>	9
4.2.2 <i>Proof of Stake (PoS)</i>	10
4.2.3 <i>Comparison of PoW & PoS</i>	10
4.2.4 <i>Byzantine Fault Tolerance (BFT)</i>	10
4.3 Regarding public and private keys	11
5. Requirements	13
5.1 Ideal Design:	13
5.1.1 <i>For vehicles</i>	13
5.1.2 <i>Road side units (RSU)</i>	14
5.1.3 <i>Consensus algorithms</i>	14
5.2 Actual Plan	14
5.2.1 <i>A competent vehicle simulation should have</i>	14
5.2.2 <i>To build a blockchain-centric system</i>	15
6. Result	16
6.1 Generation of network node data	17
6.2 The blockchain-based data receiving end	20
7. Other Attempt	30
8. Testing	32
9. Evaluation	33
10. Conclusion	36
11. Future Work	36
12. Reference	38
13. Appendices	39
13.1 Log	39
13.2 Proposal Doc	40

2. Introduction

IoT (Internet of Things) is a trend that is becoming more and more prominent nowadays. In essence, it is the connection of smart devices that people will use on a daily basis to the Internet. In such a smart city, vehicles, the most common means of transportation for modern urbanites, are also included as an important part of IoT, called IoVs (Internet of Vehicles). In an IoV, each vehicle is a node that can interact with the network. In addition to this, all the links that make up traffic, such as traffic lights, pedestrians and other vehicles, are also nodes on the IoV. The dissemination of data through such a vast network could provide a great deal of convenience for future traffic environments, for example by scanning the surroundings through each vehicle's lens sensors and transmitting them to the web to provide accurate real-time traffic conditions. But even in such a networked environment, the transmission of information is risky, and the technology of today's society cannot yet guarantee that all receiving and transmitting ends are legitimate, so there is an urgent need for a technology that can prevent information tampering to improve IoV. Blockchain is a good choice, and the composition of the distributed ledger and the consensus algorithm's constraints on nodes make it extremely difficult for hackers to tamper with the blockchain. The goal of this work is to design a blockchain-based information storage system for the IoV environment.

2.1 Motivation

Smart Cities, Internet of Things / Internet of Vehicles will gradually appear in the public eye. More and more new applications will be created and used by the public

2.2 Problems

As IoVs themselves are highly secure in terms of information, a breach of the system could have a significant and incalculable impact. For this reason, it is essential to use a secure and stable way of storing data.

2.3 Objectives

- Write the underlying blockchain logic
- Configure the system so that it has data validation capabilities
- Use the correct broadcast method to do P2P
- Automatically read and upload data to the blockchain

3. Professional Considerations and Ethics

Before embarking on this project, it is important to understand the University of Sussex's Code of Ethics and Code of Conduct and it is important that the experimental or development methods are always carried out in strict accordance with these guidelines.

In this section the ethical considerations that need to be taken into account for the project are

discussed. The conduct used to address these factors during the development of the project will also be mentioned.

The data contained in this project is generated by publicly available simulation programs and does not compromise the privacy of humans or animals in any way.

3.1 Professional consideration

This project has due regard for public health, privacy and the safety of others, and fully respects the legal rights of any third parties that may be involved. The development or testing process of this project is based on blockchain technology research and therefore does not discriminate on the basis of gender, sexual orientation, marital status, nationality, colour, race, ethnicity, religion, age or disability or any other aspect. All processes of this project, including the development through to the testing phase, will be completed by me within the limits of my professional competence. No claims will be made regarding any level of competence that I do not possess. Throughout this project I have adhered to the British Computing Society's (BCS) code of conduct, and in completing this project, which involves both research and development, I will not take credit for ideas or algorithms that I have reviewed, but will mark them up and not take credit for the work of others. I will also mark any external library or code that I have used during development to prove that I have not copied it. I respect the relevant laws and all development and testing of the blockchain system will be carried out within the limits of the law. I also accept honest criticism and guidance on this project and promise not to harm others for any reason and to reject all forms of bribery. This project will exercise professional judgement while exercising my professional duties with care as required by the relevant authorities. Information will not be divulged when using possible data for testing or debugging. In the interests of academic integrity, clear and understandable answers will be given and will not be falsified in any way.

3.2 Ethical considerations

With regard to ethical considerations, only simulated data will be used for this research project. The specific goal of this project is to develop a blockchain-based information storage system for transmitting data in a "Internet of Vehicle like" environment, which is localised and therefore no human or animal will be affected in the course of this research. With regard to the creation of the academic paper, there is also no need for human or animal involvement in its creation, and therefore no possibility of harming them. For this system, the possible test methods are network simulations, signal modelling simulations, and not real self-driving vehicle simulations. In this process, no participants would be harmed, no bribes would be paid to participants, no information would be withheld from participants and a host of other issues. All processes will be conducted in an open and transparent manner during the testing process, without the use of illegal engines or technology for digital modification or any other breach of integrity.

4. Background information

In recent times, smart devices have been gradually spreading all over the world. With industrial upgrading and the inevitable arrival of the web3 era, the number of smart devices is bound to continue to increase at a rapid pace. These industrial upgrades include, for example, the industrialisation of industries, where machines replace human labour to carry out tedious and complex tasks, or the smartisation of enterprises, which integrate the advantages of digitalisation and platform management to build server factories. Factory equipment such as this is expected to lead to a huge increase in the number of connected devices in the future, rather than just an increase in everyday devices as a result of the continuing population boom. The number of IOT-connected smart devices is predicted to reach 29.42 billion by 2030, nearly double the 15.14 billion in 2023 [1]. These devices are connected to the Internet, gather data through sensors, upload these parameters necessary for their operation to the cloud (the network side), and through a series of operations in the cloud to receive, store and analyse them, obtain valid information for the running program (e.g. instructions for the next step, etc.), which is then returned from the cloud to each specific device. This network should be available everywhere, something you can interconnect with at any time, and detect, protect, and instruct your connected devices in real time. Such a network is called the IoT (Internet of Things).

Under the IoT, one of the key subsets of applications is the IoV (Internet of Vehicles) . As mentioned above, in the IoT, each vehicle is a node that can interact with the network. The car itself has an intranet that can interact with, for example, traffic units, pedestrians, other vehicles, and so on [2]. With such interconnection, a very large network can eventually be formed, in which a huge amount of data can be disseminated and many disruptive applications can potentially be realised. These applications could include smart maps, multimedia content sharing, and so on. [3]. Such applications do bring convenience to people and add human-machine interaction in a more diverse way. But as these applications progress and the field widens, these diverse devices and distributed environments will inevitably require higher specification central servers to handle them. At the moment, the interconnection between devices is still at risk of infiltration, in other words, a series of privacy risks caused by illegal authentication and connections cannot be completely avoided.

4.1 Blockchain

4.1.1 Why we need blockchain?

Blockchain technology has therefore been brought to the fore as a solution that can be improved upon. Blockchain is a tamper-proof technology based on a distributed ledger. It essentially consists of a chain of blocks, which are formed in chronological order of appearance, and is called a blockchain. There are many such chains and they are distributed around the world in servers, which are nodes. For modifications or updates to the blockchain, there is a consensus between the nodes, and a special voting algorithm is used to ensure that the modification is approved by the majority of the nodes.

In fact, in most scenarios in this day and age, network requests between users are actually still centered on a company's database. For example, when we transfer money to someone else, our money actually goes to the bank first, which then reviews and processes it and finally sends it to the recipient. This is centralised processing, with both the transferor and the recipient of the money revolving around the bank as the hub of activity. Such a data storage and transmission model is not very risky in the current banking environment for the time being, as the multiple layers of encryption are sufficient to protect the bank's data. But imagine if the bank's back office was hacked (although this is highly unlikely in reality), then all users' accounts would be at the mercy of the hacker, and the insecurity caused by such an event could be largely solved by blockchain. The distributed nature of blockchain storage means that a blockchain has countless copies around the world, and to modify this chain requires the pulling together of nodes controlled by a consensus algorithm (usually more than half). This means that it is very difficult to tamper with the blockchain.

4.1.2 How does blockchain works?

So, how does the blockchain system work? Firstly, all individuals on a blockchain can be called a block, in which three things are stored broadly: the target data, the hash value, and the hash value of the previous block. The hash value is calculated from the data stored in the block + the previous hash value and is unique to each block. And, once the data in a block has changed, the hash value of that block will change accordingly.

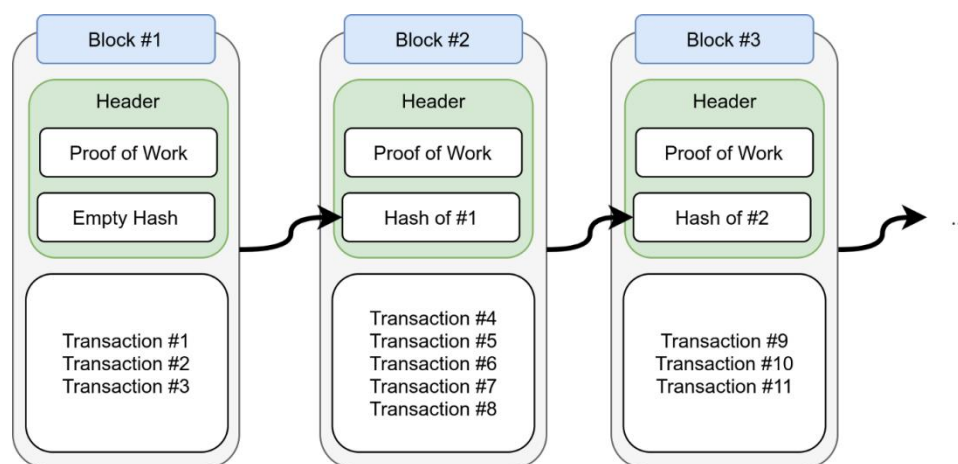


Figure1. basic components of a block (extracted from [15])

However, if a hacker needs to hack into the system, they can simply recalculate all the hashes of the altered block and all subsequent blocks and complete the calculation at high speed to restore the plausibility of the entire chain so that it will not be detected. This requires the use of consensus algorithms. The existence of consensus algorithms makes it less easy for hackers to achieve rapid data modification.

In addition, blockchain introduces the concept of P2P (Peer to Peer), a distributed network where each node (e.g. user) in the network has access to a copy of the current blockchain data. When a new block is added to the blockchain, it is broadcast to all nodes in the network, and all nodes

have the right to determine whether the block has been tampered with, and the block is only allowed to be added to the blockchain if there is a consensus among all nodes, i.e. if more than 50% of the votes on the block have been passed.

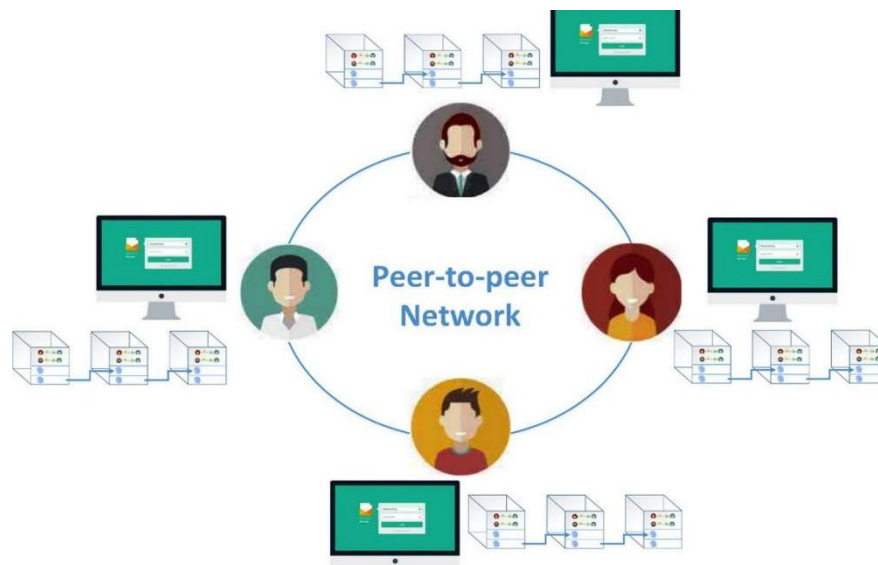


Figure2. peer to peer network (extracted from [16])

In summary, the blockchain does not curb data alteration at source, but rather makes the cost of shacking extremely large, so large that it outweighs the benefits that can be gained, thus discouraging hackers from hacking.

4.2 Consensus algorithm

There are a variety of consensus algorithms, and a few common ones are listed here as Examples [4].

4.2.1 Proof of Work (PoW)

Proof of Work (PoW) is one of the consensus algorithms. Essentially it can be understood as a difficult puzzle that the program must spend a lot of time solving in order to tamper with the hash of the block. Generally, each miner needs to compute a specific hash with a number of zeros at the beginning, the number of zeros can be flexibly adjusted according to the network-wide mining power, decreasing the number of zeros when the power is high and increasing the number of zeros when the power is low, in order to ensure that the number of blocks is one every 10 minutes, with each additional zero increasing the computational difficulty exponentially. An attacker who wants to tamper with a piece of data must not only match the timestamp and complete the proof of work for that block, but also complete the proof of work for all blocks after that block, which is an incredible amount of work unless the attacker can control 51% of the network's computing power. Therefore, it is almost impossible to compute the hash of an entire chain in a short period of time.

4.2.2 Proof of Stake (PoS)

Another algorithm is called Proof of Stake (PoS), which is a mechanism for solving inefficiencies, wasted resources and consistency problems in the network by means of equity-based bookkeeping, which simply means that whoever has more equity has the say. Compared to PoW, PoS consumes less power and handles problems more efficiently.

The main reason for all the problems with PoW is that everyone is free to be a node and each node participates in data processing through competition. The fact that a single piece of data has to be processed by so many people will certainly lead to wasted resources and inefficiency.

PoS solves this problem because PoS raises the threshold for nodes to process data by stipulating that while everyone is free to join in as a node, only nodes that meet certain conditions, such as pledging a certain amount of tokens, are eligible to become validating nodes, or candidates.

After becoming a candidate, the system will, through an algorithm, select a portion of people to be out-block nodes, and every so often, the selection will be made again. During the selection process, the algorithm will ensure that the result of the selection cannot be manipulated or predicted, thus avoiding the network being controlled by a particular node. Only by becoming a block-out node, i.e. a miner, can you participate in the processing of data and compete for the right to keep the books.

4.2.3 Comparison of PoW & PoS

PoW is where everyone can become a miner, whereas PoS is a series of filters to become a miner. The advantage of the PoS mechanism is that it solves the problems of wasted resources and inefficiencies in PoW. However, it also has some disadvantages. For example, the initial token distribution in the PoS mechanism is rather vague, and if the initial token distribution does not go down, it is difficult to form a proof of stake afterwards. Another example is that the election algorithm has the potential to be attacked, and if the attack is successful, the whole system can be manipulated.

But the biggest problem with the PoS mechanism is that it still tends to create a situation where the stronger the stronger, the easier it is for whoever has more tokens to get more tokens. Theoretically, whoever can hold 51% of the tokens can control the whole network, so it is less decentralised.

4.2.4 Byzantine Fault Tolerance (BFT)

The Byzantine Fault Tolerance (BFT) consensus algorithm is a consensus algorithm derived from the Byzantine General problem.

The Byzantine General problem is a hypothetical problem proposed by Leslie Lamport in the 1980s. Byzantium was the capital of the Eastern Roman Empire, and because the Byzantine Roman Empire was so vast and each army was so far apart, the generals had to rely on messengers to deliver their messages. In the event of war, the generals had to work out a unified plan of action. However, among these generals were traitors who wished to undermine the unified plan of action of the loyal generals by influencing the development and dissemination of the unified plan of action. Therefore, the generals must have a predetermined agreement on methods to bring all loyal generals into agreement. And a few traitors could not make the loyal

generals make the wrong plans. In other words, the essence of the Byzantine generals' problem was to find a method that would allow the generals to build consensus on a battle plan in a non-trusting environment with a version of the traitors.

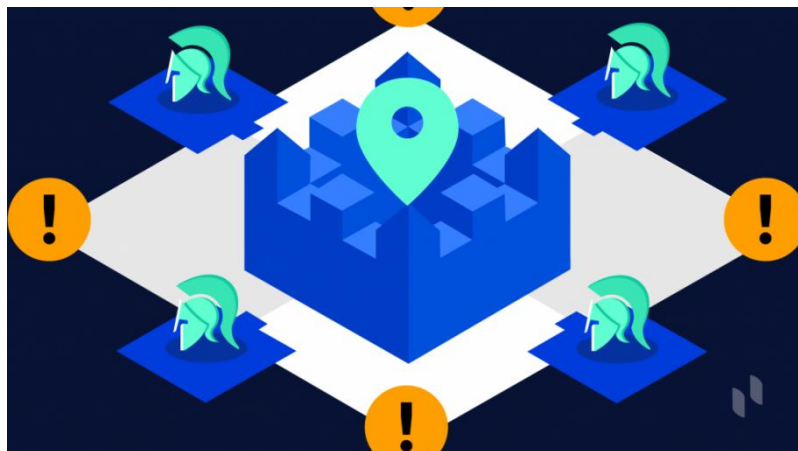


Figure3. Byzantine Fault Tolerance (extracted from [17])

The types of Byzantine algorithms can be divided into the following categories.

- Practical Byzantine fault-tolerant algorithms
- Authorized Byzantine fault-tolerant algorithms
- Federal Byzantine protocols

These algorithms differ in their specific form of implementation, but all have the characteristics of being fast, supporting high concurrency and scalable, and are usually used in private or federated chains.

4.3 Regarding public and private keys

We need to use accounts to access the blockchain system and when creating an account, a set of public and private keys need to be generated automatically as we need to use this set of keys to increase the security of our account. Both public and private keys are long strings of alphanumerics. The public key is the key that can be known by everyone, while the private key is the key that should only be known by you. Typically, we first generate a private key using a special encryption algorithm, and then use an irreversible algorithm on top of this private key to generate the public key. This is so that the public key, which can be known by others, cannot be back-propagated to obtain the private key, thus ensuring the security of our account or encrypted information.

We can encrypt a message with either a public or private key. This is mainly shown in the following diagrams:

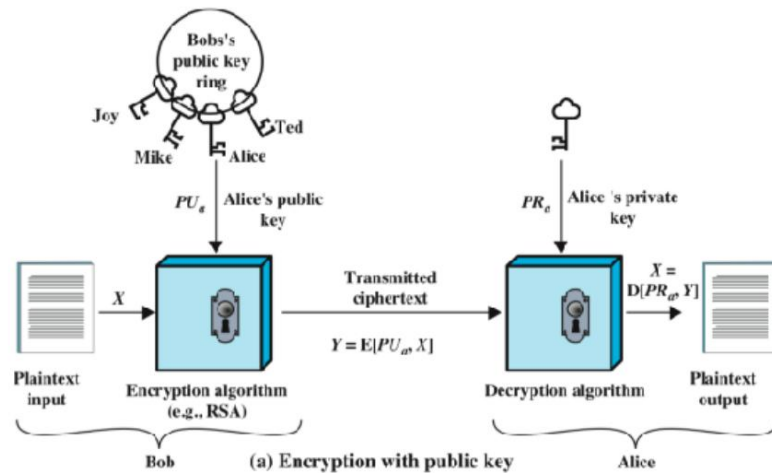


Figure4. encryption with public key (extracted from [18])

In the diagram, Bob wants to send a message to Alice, and Bob has chosen Alice's public key to encrypt the text, so in this case the text should only be decrypted by Alice's private key. If Alice's private key is not compromised, then the message should eventually be received and read only by Alice without error.

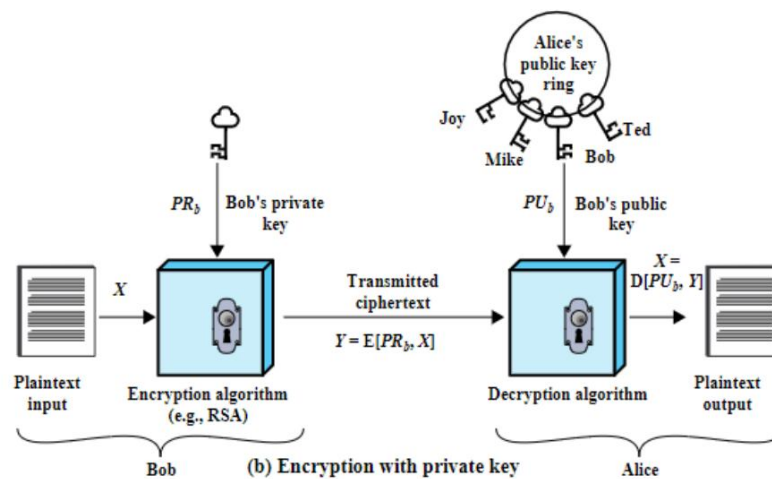


Figure5. encryption with private key (extracted from [18])

The other way is to use the private key to encrypt the text. In this example, Bob uses his private key to encrypt the text and sends the encrypted text to Alice, who uses Bob's public key to decrypt the text and obtains the plain text.

It is worth noting that if a third party intercepts the encrypted text during transmission, and this third party also has Bob's public key, then Bob's encrypted message is not secure and will be read by the third party.

To sum up, the public and private keys of the same group must be used together, i.e. the public key can only be matched with the private key of the same group for encryption or decryption operations. The first diagram is a common method for encrypting text transmissions, while the second is mainly used in the field of digital signatures.

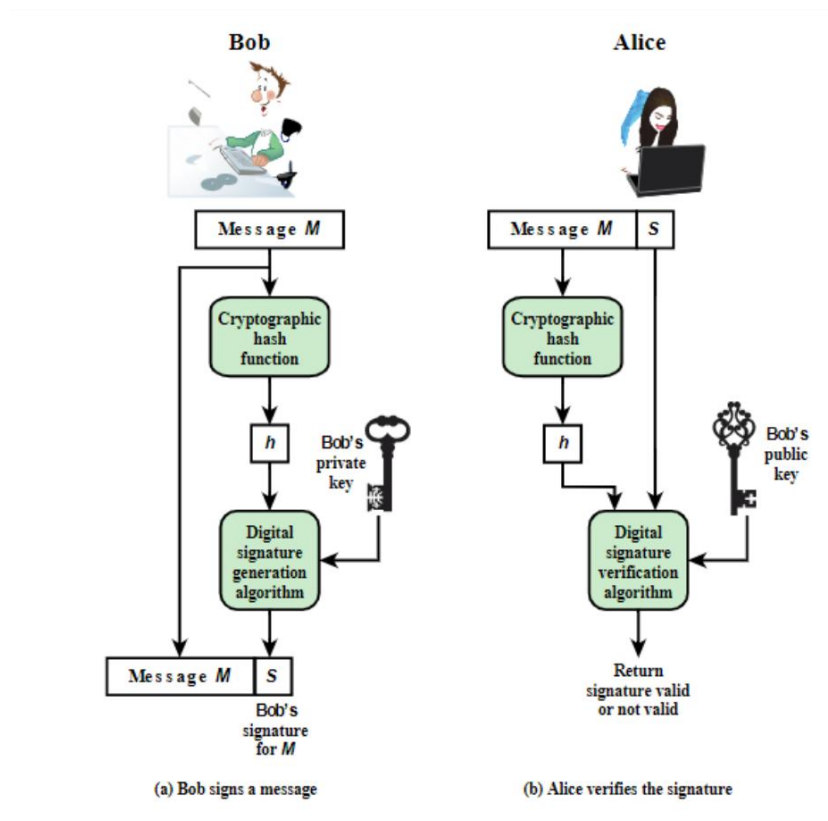


Figure6. digital signature

Digital signatures are an additional means of verifying the reliability of a text. First Bob uses the private key encryption method described above to generate a signature of the original text, such that it can only be generated by the private key and cannot be copied or imitated. After the signature has been sent with the encrypted text, it may be intercepted by a third party and the message be tampered. If the message is tampered with by a third party, then the digital signature cannot be consistent because the third party cannot have Bob's private key. Therefore, when Alice verifies the digital signature, if the result obtained by decoding with the public key matches the result obtained by the hash function, the data has not been tampered with. If not, then the data has been tampered with.

5. Requirements

5.1 Ideal Design:

5.1.1 For vehicles

Each vehicle can be understood as a node which should be equipped with sensors, transmission devices and machines with computing power which will be used to receive, analyse and share data with others. The data to be stored includes, but is not limited to, the current speed of

vehicle, cornering angles, distance to surrounding units (e.g. lane dividers, pavements, etc.). This data should be able to be propagated between vehicles within range at a high frequency over a long period of time to ensure a relatively constant balance of distance and speed between each nodes.

5.1.2 Road side units (RSU)

Not all of the above mentioned information being transmitted between vehicles is trustworthy. It is likely that a node (e.g. a vehicle) will receive multiple messages with inconsistent content, and this will require the involvement of an RSU, which, as a stable structure resident at the roadside, can be understood as a unit with a greater weight and a greater say in the trustworthiness vote. The reason for this is that roadside units, such as traffic lights/signalling devices on poles, are often wired via fibre optic cables, which are faster and less vulnerable to hack than normal wireless transmissions, so that as a more stable node, the RSU can rely on its efficient computing power to act as a manager of the traffic network in range.

5.1.3 Consensus algorithms

In such an environment, it is necessary to choose a suitable consensus algorithm to ensure the reliability of the data. Traditionally, POW presents a cracking puzzle for "miners", and such algorithms have actually proven to be reliable [5] and have been used by multiple parties [6].

However, in the context of connected vehicles that require fast information interaction, the extremely time-consuming Pow algorithm can lead to inefficiencies and even indelible traffic accidents. Another type of algorithm is PoS (Proof of Stake). Unlike the POW method - where those with more computational power have more weight - the PoS rule is that the unit with the greater equity gets more say, and this technique is low-energy and efficient. Unfortunately, it allows nodes with big stakes to accumulate stakes and win elections. If hackers take advantage of this opportunity to compromise RSUs with high stakes, the consequences can still be severe. In [7], a combination of the two methods, Pow and PoS, was proposed and this approach was considered promising. It uses PoS as the base algorithm to process data and elections in an efficient and low-energy way. However, it also uses the computational power of Pow to make it possible for temporarily underweighted RSUs to update their weights. This enables a constant flow of different RSUs to be updated to manage the network of vehicles in the region, and can go some way to avoiding the dictatorship of nodes that have been hacked into.

5.2 Actual Plan

After considering the actual time and experienced knowledge of the expertise, it was finally decided to make a demo of information transfer to enable data transfer from the simulated vehicle to the blockchain side.

5.2.1 A competent vehicle simulation should have

- There should be several nodes created.
- Each node should be initialised when it is created and have the ability to process messages (send messages, receive messages, and determine whether messages should continue to be

transmitted).

- There should be a network of nodes.
- The network should have a pre-defined task for sending messages and an upper limit for the number of messages sent and received to avoid endless running times.
- Each node on this network should be able to participate in the function of receiving and sending messages.
- There should be methods to write source code to output the data generated during the simulation of the network.
- There should be no errors reported when running the simulated network.

5.2.2 To build a blockchain-centric system

- Each block in this system should contain the basic elements required by a blockchain, i.e. block code, prior hash code, transaction specifics (not necessarily a "transaction", but a group of contents to be stored), proof (or other parameters to validate the block) and the creation time of the block.

- For each transaction, there should also be a suitable structure, which should contain, at a minimum, the data to be transferred, a digital signature and the address of which the transaction was sent.

- If you need to access the blockchain system, you need to do so via an account. Basically the most important data in the account is the public and private key. Generally, we will first generate the private key and then get the public key through a one-way irreversible algorithm. In this way the public key, which is used for dissemination on the Internet, cannot be back-propagated by others to obtain a private key that should only be known by themselves. In addition, this account should carry the current port number to differentiate the different ports during the P2P session.

- Each transaction, once verified as legitimate, should be added to a "pending zone" instead of being stored directly in the blockchain, which can be understood as a kind of cache. Transactions in the pending zone are not counted as official members of the blockchain, and in order to store transactions in the pending zone on the chain, they need to wait for the next block to be mined. After the next block is mined successfully, all transactions in the pending zone should be stored in the last block of the blockchain and the pending block should be cleared.

- Mining (PoW) is essentially asking the user's port to solve a puzzle that should take the computer a long time to respond to, for which the appropriate amount of computing power is necessary. However, in this demo we are only doing simulations, so the puzzle will not be really hard. In addition, if multiple nodes are mining a block at the same time, the first node to mine the block (i.e. solve the puzzle) should be given an incentive or some reward. At the same time, the moment he finished mining the block, the other blocks should receive a broadcast that the block has been mined successfully and automatically stop mining. Upon successful mining, the block should contain: a reward transaction for the miner's successful mining of the block, and all transactions in the "pending zone". This block should be automatically deposited at the end of the blockchain upon completion of verification.

- Peer nodes are an item that every node or port should have to store in order to let the current node know which nodes to broadcast updates with when a new transaction is added or a new block is mined.
- Because the P2P feature of blockchain systems relies on the consensus of all nodes for a unique blockchain, in other words, all nodes must have a way to acknowledge the existence of a single blockchain content. This is the consensus algorithm, which in general should be very complex and have a lot of mathematical theory behind it, but considering the size of this project, only the most basic consensus theory has been applied: the longest chain length wins, and the one with the most support wins. In addition, this account should carry the current port number to differentiate the different ports during the P2P session.
- This blockchain system should have a corresponding console-based access and html-based graphical client-facing interface.
- The blockchain system should be run without errors that would cause the process to terminate. If there is an illegal input or operation, a prompt should be popped up or an error message should be output on the console.

6. Result

Based on the outline and requirements listed above, I have completed the following plan.

First, I list the frameworks/tools to be used to configure the development environment and their version numbers:

Visual Studio Code v1.77.3

Python v3.7.16

Flask v2.2.2

Flask_cors v3.0.10

Requests v2.28.1

Pycrypto v2.6.1

Tinydb v4.7.1

OMNET++ v5.6.2

For this project I decided to use Visual Studio Code (VS Code), an open source lightweight code editor, which I thought would suit my needs as it is fast to start, easy to install plugins and has a neat interface. For building the site I introduced the Flask framework, which like VS Code is just as simple and fast, a micro framework with no default database, form validator. The Flask_cors

plugin was also introduced to address the need to share resources across origin that I would encounter in the project. Since Flask does not have a default database, I introduced TinyDB, a lightweight database written in pure Python with no external dependencies. There are no complex and lengthy functions, data can be written to JSON files via a simple api, and the database supports Python v3.6+, which is a good match for the version of Python I am using. For the HTTP request methods that need to be used in the data broadcast, I use the Requests library to assist, which makes it easy to send HTTP requests to the website. For the most important part of encryption (e.g. public and private keys) I chose the Pycrypto library, which provides many of the algorithms I would use for public key encryption, one-way encryption and so on.

In addition, the OMNET++ IDE is used to simulate basic network node communication. It is a modular component-based open network simulation platform that is very powerful in all aspects.

6.1 Generation of network node data

For the simulation of network nodes, the TicToc example actually given officially by OMNET++ is used. I have added the initialisation of documents and the writing of messages to it. In this simulation, a set of networks with a tree topology are created and automatically send and receive messages to and from each other. Whenever a message is generated, one among all nodes is randomly selected as the destination and the message is sent. After the message has been sent, the nodes in the route will randomly pass this message to neighbouring nodes until the message reaches the set destination.

During the simulation, the majority of messages are passed multiple times due to the random nature of the destination and the target of the pass. We use the hopCountVector to keep track of the number of times each message is passed, and eventually store the data in a text file to be read by the blockchain system.

```

42=class Txc15 : public cSimpleModule
43 {
44     private:
45         long numSent;
46         long numReceived;
47         cLongHistogram hopCountStats;
48         cOutVector hopCountVector;
49         string savedInFile = "dataOutput\\dataTxc15.txt";
50
51     protected:
52         virtual TicTocMsg15 *generateMessage();
53         virtual void forwardMessage(TicTocMsg15 *msg);
54         virtual void initialize() override;
55         virtual void handleMessage(cMessage *msg) override;
56
57         // The finish() function is called by OMNeT++ at the end of the simulation:
58         virtual void finish() override;
59 };
60
61 Define_Module(Txc15);

```

```

63 void Txc15::initialize()
64 {
65     // Initialize variables
66     numSent = 0;
67     numReceived = 0;
68     WATCH(numSent);
69     WATCH(numReceived);
70
71     hopCountStats.setName("hopCountStats");
72     hopCountStats.setRangeAutoUpper(0, 10, 1.5);
73     hopCountVector.setName("HopCount");
74
75     // Module 0 sends the first message
76     if (getIndex() == 0) {
77         // Boot the process scheduling the initial message as a self-message.
78         TicTocMsg15 *msg = generateMessage();
79         scheduleAt(0.0, msg);
80     }
81
82     // clear the output file if exists
83     struct stat buffer;
84     if ( stat(savedInFile.c_str(), &buffer) == 0 ){
85         ofstream file_writer(savedInFile, ios_base::out);
86         file_writer.close();
87     }
88 }
89 }
90

```

Figure7. tictoc configuration file

When initialising, a message is generated after the necessary variables have been set, after which the target file (path stored in savedInFile) is checked to see if it exists. If it does exist, the file would be cleared. This operation is done to facilitate the repetition of the simulation.

```

134 TicTocMsg15 *Txc15::generateMessage()
135 {
136     // Produce source and destination addresses.
137     int src = getIndex();
138     int n = getVectorSize();
139     int dest = intuniform(0, n-2);
140     if (dest >= src)
141         dest++;
142
143     char msgname[20];
144     sprintf(msgname, "tic-%d-to-%d", src, dest);
145
146     // Create message object and set source and destination field.
147     TicTocMsg15 *msg = new TicTocMsg15(msgname);
148     msg->setSource(src);
149     msg->setDestination(dest);
150     return msg;
151 }
152

```

Figure8. generateMessage function

In the generateMessage() function, a random destination is generated that is not equal to the origin one, and then a message object is created and sent.

```

152
153 void Txc15::forwardMessage(TicTocMsg15 *msg)
154 {
155     // Increment hop count.
156     msg->setHopCount(msg->getHopCount()+1);
157
158     // Same routing as before: random gate.
159     int n = gateSize("gate");
160     int k = intuniform(0, n-1);
161
162     EV << "Forwarding message " << msg << " on gate[" << k << "]\n";
163     send(msg, "gate$o", k);
164 }
165

```

Figure9. forwardMessage function

In forwardMessage(), the hop_count is added by one and the next destination of the message is determined randomly.

```

91 void Txc15::handleMessage(cMessage *msg)
92 {
93     TicTocMsg15 *ttmsg = check_and_cast<TicTocMsg15 *>(msg);
94
95     if (ttmsg->getDestination() == getIndex()) {
96         // Message arrived
97         int hopcount = ttmsg->getHopCount();
98         EV << "Message " << ttmsg << " arrived after " << hopcount << " hops.\n";
99         bubble("ARRIVED, starting new one!");
100
101         // update statistics.
102         numReceived++;
103         hopCountVector.record(hopcount);
104         hopCountStats.collect(hopcount);
105
106         delete ttmsg;
107
108         // Generate another one.
109         EV << "Generating another message: ";
110         TicTocMsg15 *newmsg = generateMessage();
111         EV << newmsg << endl;
112         forwardMessage(newmsg);
113         numSent++;
114
115         ofstream oFile;
116         oFile.open(savedInFile, ios::app);
117         oFile << hopCountVector.getValuesStored() << "\n";
118         oFile.close();
119
120         // stop when tic[0] has received the msg for 10 times
121         if (numReceived >= 10) {
122             endSimulation();
123         }
124
125         // buffer
126         std::this_thread::sleep_for(std::chrono::milliseconds(1000));
127     }
128     else {
129         // We need to forward the message.
130         forwardMessage(ttmsg);
131     }
132 }
133

```

Figure10. handleMessage function

Whenever a node receives a message, handleMessage() is called. First determines if destination has been reached, if the current node is not destination, then forwards the message; if destination has been reached, then sends an arrival acknowledgement and generates a new message. Store the parameter hop_count in a txt file. I set it to automatically end the simulation when tic[0] has received the message 10 times. The sleep_for in the last line is to prevent the simulation from generating data too fast and overloading the cryptographic side of the blockchain.

```

1 simple Txc15
2 {
3     parameters:
4         @display("i=block/routing");
5     gates:
6         inout gate[];
7 }
8
9 network Tictoc15
10 {
11     types:
12         channel Channel extends ned.DelayChannel {
13             delay = 100ms;
14         }
15     submodules:
16         tic[6]: Txc15;
17     connections:
18         tic[0].gate++ <--> Channel <--> tic[1].gate++;
19         tic[1].gate++ <--> Channel <--> tic[2].gate++;
20         tic[1].gate++ <--> Channel <--> tic[4].gate++;
21         tic[3].gate++ <--> Channel <--> tic[4].gate++;
22         tic[4].gate++ <--> Channel <--> tic[5].gate++;
23 }
24

```

Figure11. network defined in the .ned file

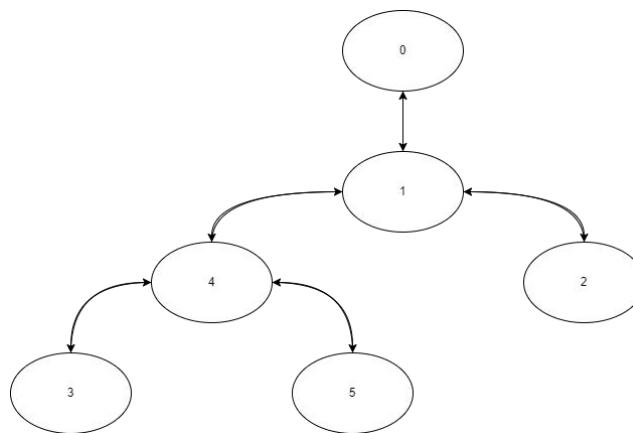


Figure12. this network can be seen as a tree topology

The entire network is a tree topology, where neighbouring nodes can transmit information to each other.

6.2 The blockchain-based data receiving end

(The proposed blockchain system is based on: [8])



Figure13. project structure

In the above diagram, the db path stores the json database files generated by tinydb; the ui path stores the html files about the web interface design; the utility path stores the frequently used tool files, and the other files are the object-oriented files that make up the python-based local blockchain.

```

def generate_keys(self):
    private_key = RSA.generate(1024, Crypto.Random.new().read)
    public_key = private_key.publickey()
    return (binascii.hexlify(private_key.exportKey(format='DER')).decode('ascii'),
            binascii.hexlify(public_key.exportKey(format='DER')).decode('ascii'))

def create_keys(self):
    private_key, public_key = self.generate_keys()
    self.private_key = private_key
    self.public_key = public_key
  
```

Figure14. key generation

For a system, users often need to register an account to access content within the site. This system is no exception. You need to create a public key and a private key pair as your address to access the site, otherwise most of the functionality will not be available. In each account (called a wallet in this case) there is a public key, a private key and a port number. At the beginning, the public and private keys are set to None by default, and only after the user has used the "create

keys" method will a set of public and private keys be generated, while the port number is entered by the user when creating the port and is passed into a set to avoid duplication.

In the diagram, the key is generated using the RSA algorithm, which is an asymmetric encryption algorithm. It uses the public key for encryption and the private key for decryption. This algorithm is called asymmetric because the private and public keys are two different strings. 1024 bit size is chosen and the strong secret key is generated by passing `Crypto.Random.new().read` into the `randfunc` parameter. thereafter the strong secret key is used in the private `public_key()` to generate the public key to ensure that the private key is not back-propagated by others. Finally return the keys in DER format.

```

5  class Transaction(Printable):
6      """
7      Use the form and concept of transaction, but not with money transfer
8      """
9
10     def __init__(self, dataOwner, signature, hop_count):
11         """
12         self.dataOwner:        your public key
13         self.signature:        digital signature
14         self.hop_count:        data we want to upload to the blockchain
15         """
16         self.dataOwner = dataOwner
17         self.signature = signature
18         self.hop_count = hop_count
19

```

Figure15. Transaction class

In the design of this system, all file transfers are based on "transactions", so a "transaction class" has been created, but unlike traditional ones, this transaction does not include fees such as fuel costs for the transfer of information. In the same way that the wallet class (user accounts) are not allocated Balance or anything similar to money. In this project, we are only concentrating on the transmission of information. Therefore, in the transaction class, the only variables included are `dataOwner` (public key), `digital signature` and `hop_count` (the data to be transferred)

```

def sign_transaction(self, dataOwner, hop_count):
    """
    Sign a transaction and return the signature

    dataOwner:        your public key
    hop_count:        data we want to upload to the blockchain
    """

    signer = PKCS1_v1_5.new(RSA.importKey(binascii.unhexlify(self.private_key)))
    h = SHA256.new((str(dataOwner) + str(hop_count)).encode('utf8'))
    signature = signer.sign(h)
    return binascii.hexlify(signature).decode('ascii')

```

Figure16. sign_transaction function

When signing a digital signature we will call the `sign_transaction` function in the wallet class to sign it. To ensure the uniqueness of the digital signature, we need to combine all the parameters

in the transaction object except the “signature” to generate the digital signature. First we generate the signer using the PKCS1_V1_5.new() function from Pycrypto, passing in our own private key, then we hash the dataOwner and hop_count parameters in the transaction using SHA256. Finally, use the sign() function on the signer, passing in the hash value "h" just generated, to complete the signature.

```

4
5 class Block(Printable):
6     """
7     Basic block for blockchain
8     """
9
10    def __init__(self, index, previous_hash, transactions, proof, time=time()):
11        self.index = index
12        self.previous_hash = previous_hash
13        self.transactions = transactions
14        self.proof = proof
15        self.timestamp = time
16
17

```

Figure17. Block class

This blockchain contains the basic properties of a block, index is the block number, previous hash is the hash of the previous block, transaction is the main content carried by the block, proof is the "proof of work" needed to mine the block, and timestamp is the time when the block was created. The previous hash is used to link blocks together in a tight sequence because of its irreversibility.

```

8
9 @staticmethod
10 def valid_proof(transactions, last_hash, proof):
11     """
12     Validate a proof of work number and see if it solves the puzzle algorithm
13
14     transactions:    The transactions of the current block
15     last_hash:       Previous block's hash
16     proof:           The proof number
17     """
18     guess = (str([tx.to_ordered_dict() for tx in transactions]) + str(last_hash) + str(proof)).encode()
19     guess_hash = hash_string_256(guess)
20     return guess_hash[0:2] == '00'
21
144
145 def proof_of_work(self):
146     """ to get a string starts with 00 """
147     last_block = self.__chain[-1]
148     last_hash = hash_block(last_block)
149     proof = 0
150
151     while not Verification.valid_proof(self.__open_transactions, last_hash, proof):
152         proof += 1
153     return proof
154
155

```

Figure18. validation methods

The "proof of work" is computed one at a time by using the valid_proof() method in verification.py. As you can see from the diagram, we hash it by passing open_transaction, last_hash and proof into valid_proof(), with the goal of getting a number of hashes starting with "00". Proof will add one to each failed attempt, and the next guess will be made through this

change until a legal hash is obtained. Once the legal hash is obtained, the miner's mining task is complete. The difficulty of such a puzzle increases with the 0 of hashes starting with. For example, the goal of getting a hash starting with "000" is more difficult than getting a hash starting with "00". Of course, it doesn't have to be 00, but any number, such as starting with "11" or "22", and the principle is the same. The key point is that with each additional prefix, the difficulty and time spent increases exponentially.

```

16
17 class Blockchain:
18     """
19     Basic blockchain
20
21     genesis_block:        the first block for all the blockchains
22     self.__chain:         blockchain
23     self.__open_transactions: the transactions that still waiting for writing into the blockchain
24     self.__peer_nodes:    nodes that can interact with
25     """
26
27     def __init__(self, public_key, node_id):
28         # the first block in the chain
29         genesis_block = Block(0, '', [], 100, 0)
30         # blockchain
31         self.__chain = [genesis_block]
32         # pending
33         self.__open_transactions = []
34         # peer nodes
35         self.__peer_nodes = set()
36
37         # load blockchain
38         self.public_key = public_key
39         self.node_id = node_id
40         self.resolve_conflicts = False
41         self.load_data()
42

```

Figure19. Blockchain class

The main components of a blockchain can be seen in the diagram above. For the first block in the chain, it cannot be generated in the normal way because of the feature of the blockchain's previous_hash generation which requires the content of the previous block, so we manually create a block with empty content in __init__ as the genesis block for the whole blockchain. For open_transaction, it is essentially a pending zone that is not formally a part of the chain. The contents of the pending zone will not be formally transferred to the chain until a block is successfully mined. Peer_nodes is fundamental to the system's implementation of P2P, and the system will broadcast nodes one by one according to the nodes stored in this variable. Since nodes cannot be duplicated, the peer_nodes is defined in the set format.

The Blockchain object also stores the public key, node id (port number) for subsequent calls and resolve_conflicts is a variable that determines whether the blockchain is consistent between nodes. Finally, all the data stored in the database (blockchain, wallet) is loaded into the blockchain object.


```

156 def add_transaction(self, dataOwner, signature, hop_count, is_receiving=False):
157     """
158     Append the transaction to the open_transaction list
159
160     dataOwner:          your public key
161     signature:          digital signature
162     hop_count:          data we need to upload
163     is_receiving:       check if this calling is a broadcast from other nodes
164     """
165
166     transaction = Transaction(dataOwner, signature, hop_count)
167     if Verification.verify_transaction(transaction):
168         self.__open_transactions.append(transaction)
169         self.save_data(save_opentx=True)
170
171     # Broadcasting
172     if not is_receiving:
173         for node in self.__peer_nodes:
174             url = 'http://{}/broadcast-transaction'.format(node)
175             try:
176                 response = requests.post(url, json={
177                     'dataOwner': dataOwner,
178                     'signature': signature,
179                     'hop_count': hop_count
180                 })
181                 if response.status_code == 400 or response.status_code == 500:
182                     print('Transaction declined, needs resolving')
183                     return False
184             except requests.exceptions.ConnectionError:
185                 print('Error occurred when Broadcasting Transaction')
186                 continue
187         return True
188     return False

```

```

@staticmethod
def verify_transaction(transaction, check_upload=True):
    """
    Check that the upload is not empty and transaction is valid
    """

    if check_upload:
        return transaction.hop_count != None and Wallet.verify_transaction(transaction)
    else:
        return Wallet.verify_transaction(transaction)

```

```

@staticmethod
def verify_transaction(transaction):
    """
    Verify the signature of a transaction.
    """

    public_key = RSA.importKey(binascii.unhexlify(transaction.dataOwner))
    verifier = PKCS1_v1_5.new(public_key)
    h = SHA256.new((str(transaction.dataOwner) + str(transaction.hop_count)).encode('utf8'))
    return verifier.verify(h, binascii.unhexlify(transaction.signature))

```

Figure20. add_transaction and functions related

The add_transaction method creates a new transaction object with the received transaction content parameters and then immediately performs the validation of the transaction. The validation here is divided into the validation of the data and the validation of the wallet: for the data (hop_count), we check to ensure that it is not empty, while the overall check of the wallet is actually similar to the process of sign_transaction, except that the public key is used to verify the signature, and if the structure is true, it means that no document has been modified by a third party during the transmission process. If the structure is true, it means that the document has not been modified by a third party during transmission. If not, it means that there is a crisis of information being intercepted.

After this verification, the transaction is stored in the open_transaction variable and written to the database. It is then broadcast to all peer nodes. This is controlled by the is_receiving variable, which is False by default, so not False = True, so it will be broadcasted by default. And when it is broadcasted to another node, it will be passed the is_receiving = True parameter when it calls the add_transaction method, so it will not broadcast again in order to avoid the result of an infinite loop.

```

190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227

def mine_block(self):
    """
    Add a new block to the current chain,
    The tx in open_transaction will be added to this block
    """

    if self.public_key == None:
        return None

    hashed_block = hash_block(self.__chain[-1])
    proof = self.proof_of_work()
    copied_transactions = self.__open_transactions[:]
    for tx in copied_transactions:
        if not Wallet.verify_transaction(tx):
            return None

    block = Block(len(self.__chain), hashed_block, copied_transactions, proof)
    self.__chain.append(block)
    self.__open_transactions = []
    self.save_data(save_chain=True, save_opentx=True)

    # Broadcasting
    for node in self.__peer_nodes:
        url = 'http://{}/broadcast-block'.format(node)
        converted_block = block.__dict__.copy()
        converted_block['transactions'] = [tx.__dict__ for tx in converted_block['transactions']]
        try:
            response = requests.post(url, json={'block': converted_block})
            if response.status_code == 400 or response.status_code == 500:
                print('Block declined, needs resolving')
            if response.status_code == 409:
                self.resolve_conflicts = True
        except requests.exceptions.ConnectionError:
            print('Error occurred when Broadcasting Block')
            continue
    return block

```

Figure21. mine_block function

In the mine_block method, the user must have the public key, i.e. the address, in order to mine. Therefore, if the current public key is not loaded, the request is returned as None. After this, a digital signature check is performed to prevent tampering with the information. After passing these checks, the block is considered to be mined and valid, so a new block is created and added to the end of the blockchain. At the same time all transactions in the open transaction should be cleared, as the transactions there should already be included in the new block.

Once this data has been updated to the database, the corresponding broadcast process will take place, just like the previous add_transaction(). The function broadcasts the nodes stored in peer_nodes one by one. However, unlike add_transaction(), there is no is_receiving parameter in mine_block to help check the broadcast status. This is because after the requests.post() method is called, the corresponding html method will call another function in the blockchain: add_block()

```

165
166 @app.route('/broadcast-block', methods=['POST'])
167 def broadcast_block():
168     values = request.get_json()
169     if not values:
170         response = {'message': 'No data found.'}
171         return jsonify(response), 400
172     if 'block' not in values:
173         response = {'message': 'Some data is missing.'}
174         return jsonify(response), 400
175     block = values['block']
176     if block['index'] == blockchain.chain[-1].index + 1:
177         if blockchain.add_block(block):
178             response = {'message': 'Successfully added block'}
179             return jsonify(response), 201
180         else:
181             response = {'message': 'Block seems invalid.'}
182             return jsonify(response), 409
183     elif block['index'] > blockchain.chain[-1].index:
184         response = {'message': 'Blockchain seems to differ from local blockchain.'}
185         blockchain.resolve_conflicts = True
186         return jsonify(response), 200
187     else:
188         response = {'message': 'Blockchain seems to be shorter, block not added'}
189         return jsonify(response), 409

```

```

228
229 def add_block(self, block):
230     """ Create a new block, used for broadcasting"""
231
232     transactions = [Transaction(
233         tx['dataOwner'],
234         tx['signature'],
235         tx['hop_count']
236     ) for tx in block['transactions']]
237
238     proof_is_valid = Verification.valid_proof(transactions, block['previous_hash'], block['proof'])
239     hashes_match = hash_block(self.__chain[-1]) == block['previous_hash']
240     if not proof_is_valid or not hashes_match:
241         return False
242
243     converted_block = Block(block['index'],
244                             block['previous_hash'],
245                             transactions,
246                             block['proof'],
247                             block['timestamp'])
248
249     self.__chain.append(converted_block)
250     stored_transactions = self.__open_transactions[:]
251
252     # check the open transaction when broadcasting
253     for itx in block['transactions']:
254         for opentx in stored_transactions:
255             if opentx.dataOwner == itx['dataOwner'] and opentx.signature == itx['signature'] and opentx.hop
256                 try:
257                     self.__open_transactions.remove(opentx)
258                 except ValueError:
259                     print('Opentx not found, maybe it is already removed')
260     self.save_data(save_chain=True, save_opentx=True)
261     return True
262

```

Figure22 broadcast-related functions

Following on from the previous chapter, the broadcast is started in the mine_block function. After the html method is called, the thread enters the broadcast_block function in node.py, and after some checks for blocks, the function calls blockchain.add_block() to broadcast to all nodes. The same series of checks for "proof of work", "blocks" will be performed during the broadcast, and will not be repeated here. The block is added to the chain after the checks have been made.

After that, for open_transactions on other nodes, we use a for loop to check and remove all the transactions added to the new block from the open_transactions. This is the normal route for mining new blocks and broadcasting them.

```

264 def resolve(self):
265     """ resolve the blockchain conflicts, the longer one wins """
266
267     winner_chain = self.__chain
268     replace = False
269     for node in self.__peer_nodes:
270         url = 'http://{}/chain'.format(node)
271         try:
272             response = requests.get(url)
273             node_chain = response.json()
274             node_chain = [Block(
275                 block['index'],
276                 block['previous_hash'],
277                 [Transaction(
278                     tx['dataOwner'],
279                     tx['signature'],
280                     tx['hop_count']
281                 ) for tx in block['transactions']],
282                 block['proof'],
283                 block['timestamp']
284             ) for block in node_chain]
285
286             node_chain_length = len(node_chain)
287             local_chain_length = len(winner_chain)
288
289             # replace the BC if the received one is longer
290             if node_chain_length > local_chain_length and Verification.verify_chain(node_chain):
291                 winner_chain = node_chain
292                 replace = True
293         except requests.exceptions.ConnectionError:
294             continue
295     self.resolve_conflicts = False
296     self.__chain = winner_chain
297     if replace:
298         self.__open_transactions = []
299     self.save_data(save_chain=True, save_opentx=True)
300     return replace
301
302

```

Figure23. conflicts resolution function

In addition to the regular blockchain system functionality, we should also consider the handling of errors when they occur. In the resolve() function, an attempt is made to solve the problem of inconsistent blockchain progress across nodes. First a variable winner_chain is set, with the default value being the current node's chain, and then the peer nodes' chains are fetched one by one and compared to their lengths. Here only the length of the chain is looked at, and the node with the longer chain wins. This means that if the chain of this node is shorter than the chain of peer nodes, then the local one will be updated. Conversely, the local one will not be updated.

```

273
274 # local file check
275 @app.route('/file-check', methods=['POST'])
276 def timed_check():
277     global read_from, port, loop_count
278     print('port is ', port)
279     file_directory = "../tictoc/dataOutput/dataTxc15.txt"
280
281     while loop_count <= 100:
282         try:
283             with open(file_directory, mode='r') as f:
284                 file_content = f.readlines()[read_from:]
285                 read_from += len(file_content)
286                 print('Pointer is at: ', read_from)
287
288                 if len(file_content) != 0:
289                     for row in file_content:
290                         print('looping ... ')
291                         if add_tx_backend(row):
292                             print('Successfully added tx from backend')
293                         else:
294                             print('Error occurred in add_tx_backend')
295                             break
296                     print('Out of loop')
297                     mine_backend()
298                     print('===== MINED A BLOCK SUCCESSFULLY =====')
299         except (IOError, IndexError):
300             print("An error occurred during time_check !!")
301             return False
302
303         time.sleep(1)
304         loop_count += 1
305         print('IT\'S LOOP ', loop_count, 'RN')
306
307     response = {
308         'message': 'Successfully getc checked'
309     }
310     return jsonify(response), 200
311
312
313 def add_tx_backend(hop_count):
314     if wallet.public_key == None or hop_count == None:
315         return False
316     signature = wallet.sign_transaction(wallet.public_key, hop_count)
317     success = blockchain.add_transaction(wallet.public_key, signature, hop_count)
318     if success:
319         return True
320     return False
321
322
323 def mine_backend():
324     try:
325         blockchain.mine_block()
326     except TypeError or TimeoutError:
327         raise Exception("An error occurred when calling mine_backend !!")
328
329

```

Figure24. functions about receiving data

For the simulation data generated from OMNET++ above, I wrote a file receiving function in the python blockchain project, time_check() , in which the function checks the specified file and performs 100 loops. During the run of this function, running the above simulation and writing the data continuously to dataTxc15.txt, the corresponding fileIO check will be triggered and the data will be extracted one by one and added to the blockchain in normal procedures. Additional add_transaction() and mine() methods are written to distinguish between manually entered data and data obtained by scanning the file.

7. Other Attempt

After the narrative of the completed python blockchain demo implementation, the process of trying it out on other paths should be mentioned.

Frameworks/tools used and their version numbers:

Truffle v5.7.5

Ganache v7.7.3

Solidity v0.5.16 (solc-js)

Node v18.12.1

Web3.js v1.8.2

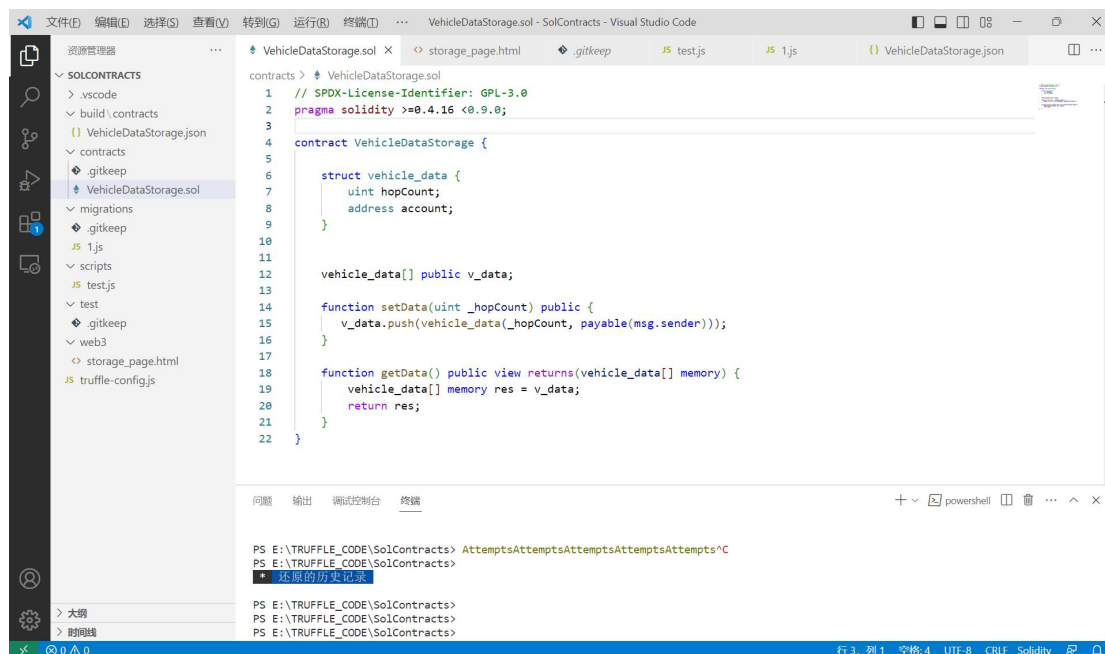
Veins v5.2

Inet v4.2.2

SUMO v1.15.0

OMNET++ v5.6.2

I have experimented with the call and use of existing Ethereum frameworks. On the smart contract side, I used the Ethereum Truffle framework to write a simple smart contract for storing data using the Solidity language. After migration, a console-based test script file was written for basic testing.



```
contracts > VehicleDataStorage.sol
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.4.16 <0.9.0;
3
4 contract VehicleDataStorage {
5
6     struct vehicle_data {
7         uint hopCount;
8         address account;
9     }
10
11     vehicle_data[] public v_data;
12
13     function setData(uint _hopCount) public {
14         v_data.push(vehicle_data(_hopCount, payable(msg.sender)));
15     }
16
17     function getData() public view returns(vehicle_data[] memory) {
18         vehicle_data[] memory res = v_data;
19         return res;
20     }
21 }
22 }
```

```
PS E:\TRUFFLE_CODE\SolContracts> AttemptsAttemptsAttemptsAttemptsAttempts^C
PS E:\TRUFFLE_CODE\SolContracts>
PS E:\TRUFFLE_CODE\SolContracts>
PS E:\TRUFFLE_CODE\SolContracts>
```

Figure25. Smart Contract demo

ACCOUNTS

BLOCKS

TRANSACTIONS

CONTRACTS

EVENTS

LOGS

SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK

5

GAS PRICE

20000000000

GAS LIMIT

6721975

HARDFORK

MUIRGLACIER

NETWORK ID

5777

RPC SERVER

HTTP://127.0.0.1:7545

MINING STATUS

AUTOMINING

WORKSPACE

USED-RELATION

SWITCH

MNEMONIC

deny detail rent crouch moon ceiling topple hollow swarm slim rail glow

HD PATH

m/44'/60'/0'/0/account_index

ADDRESS

0x8A2604AB6640901f7D389a28FbEf8062692F2594

BALANCE

99.98 ETH

TX COUNT

5

INDEX

0

ADDRESS

0x919FA2555b4e07e279A117aD1A4ff03DB1cfc1D7

BALANCE

100.00 ETH

TX COUNT

0

INDEX

1

ADDRESS

0x44aE8dFfdbb32a3f8A3cBB30BfE31D1f7590f62f

BALANCE

100.00 ETH

TX COUNT

0

INDEX

2

ADDRESS

0x1F10b25c6DE8F9dA6779396F50F3494474D3AD28

BALANCE

100.00 ETH

TX COUNT

0

INDEX

3

ADDRESS

0xc2043b585E219921bea4c3d4Fff4104Eb7739d59

BALANCE

100.00 ETH

TX COUNT

0

INDEX

4

ADDRESS

0xb8A2d380a2d38784484a5972F15B103DB17AD17F

BALANCE

100.00 ETH

TX COUNT

0

INDEX

5

ADDRESS

BALANCE

TX COUNT

INDEX

Figure26. Ganache

For the client side, I used Ganache to set up a local network and ran it. For the web-based client validation tool and wallet, I chose Metamask, and after registering I manually added the Ganache local network and the corresponding account that I had just generated. Then I wrote the basic html file and let this html layout connect to my solidity via the contract and address abi, which is written in JSON and obtained from 'migration' command. And the external library of Web3.js can be used to retrieve the user's login status, the user's public key (address), and the current balance of that address, etc. A preliminary ethereum information storage interface was built like this.

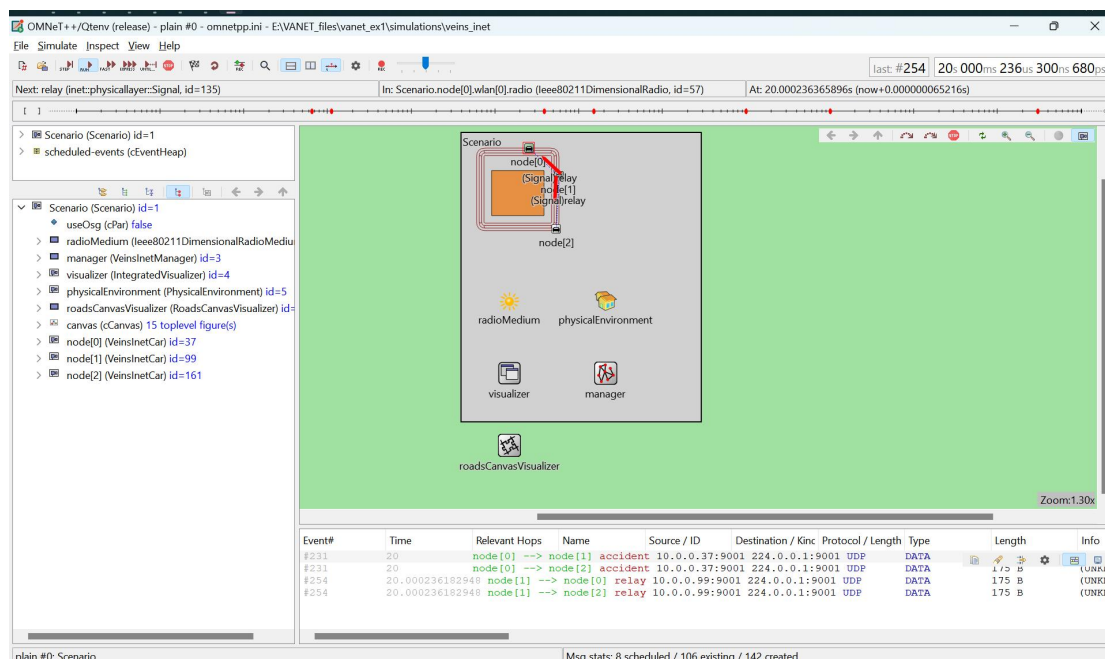


Figure27. VeinsNet simulation

For the network simulation side, I also used OMNET++ IDE, an component-based network simulator, together with Veins, Inet and SUMO. SUMO is a simulator for road traffic, Veins is a vehicle network simulation framework based on the OMNET++ and SUMO, while Inet is used to simulate mobile networks. I have simulated a simple vehicle network by retrieving the example files included in Veins and Inet. During the simulation, three vehicles were generated on a circular road, sending signals to each other several times while driving, and then retiring one by one.

With regard to the transmission of information, since the c++ data from the OMNET++ network simulation could not be converted directly with the network side built with solidity, and no relevant c++ to solidity external library was found on the web, an external medium had to be found to act as a relay station for the information transmission. At one point I tried the IPFS (InterPlanetary File System) tool for storing and reviewing information. Unfortunately, IPFS does not provide an official lib file for c++, so I had to compile it myself via CMake based on the given header file. But, during the compilation process it kept reporting errors that could not get the required files. Besides, it was subsequently discovered that IPFS did not have an interface for Solidity, so this method was finally abandoned.

8. Testing

In this section, I will complete a step-by-step demonstration of data collection plus transfer to an empty blockchain. The exact detailed steps will be listed. The test results will then be illustrated in tabular form.

Steps of storing simulation data

- First open two terminals in the blockchain_loV directory and type "python node.py" and "python node.py -p 5001" respectively. Start localhost:5000 and localhost:5001 with this.
- Check the Load Wallet function, access should be denied and the database should be empty. Check the Load Blockchain function, there should be one and only one Genesis block.
- Go to the network page and add localhost:5000 and localhost:5001 to each other as peer nodes.
- Create new wallet for each localhost. That should be done successfully and the public key and private key will be shown on the page.
- Open OMNET++ and open the txc15.cc. Click the "run" button in the tool bar. The simulation window should pop up smoothly. Don't run the simulation, stay here for now.
- Back to the localhost, choose whatever port 5000 or 5001, click the black button "Scanning Local File" (There should be only on port running this scanning function). After that, go to the

OMNET simulation window, and run the simulation. The result should be written in a text file, and read by blockchain system, finally stored in the chain database. Hint messages should also be outputted in the terminal. In total 40 transactions should be stored in the blockchain and successfully broadcast.

No.	Test content	Description	Result
1	Load the server	Run the servers	Success
2	Wallet	Cannot load it while database is empty. Can load it after created one.	Success
3	Blockchain	Should only have a genesis block when the database is empty. After a block has been mined, it should be updated with it.	Success
4	Open transaction	Storing a data manually, it should be stored in the open transactions	Success
5	Mine Block	A block should be added in the blockchain, it should contains what we have in open transaction. The open transaction should be cleared simultaneously	Success
6	Peer node	Can add a node, delete a node, load nodes freely	Success
7	Resolve Conflicts	When local chain is shorter than the peer node's one, the local chain should be covered	Success
8	Scanning Local File	Hint messages should keep updating in the terminal until the end.	Success

9. Evaluation

The project can be broadly divided into two parts in terms of implementation: the blockchain-based server side, and the data sending side for network simulation.

For the blockchain system, this project uses python and a number of lightweight tools to implement a blockchain system with basic functionality. In this system, you can create wallets, send transactions, mine blocks, fix conflicts, and add/remove peer nodes.

The time taken to encrypt a transaction, or to create a block, can be used as an evaluation metric for performance when the system is in operation.

When encrypting transactions, the digital signature scheme is used as a verifier and the SHA256 algorithm is used to obtain a hash number of the transaction data. When mining blocks, SHA256 is also used as the encryption algorithm, except that all transactions for the entire block are

processed.

```
127.0.0.1 - - [27/Apr/2023 10:23:39] "GET /nodes HTTP/1.1" 200 -
127.0.0.1 - - [27/Apr/2023 10:23:49] "GET / HTTP/1.1" 304 -
127.0.0.1 - - [27/Apr/2023 10:23:49] "GET /wallet HTTP/1.1" 201 -
127.0.0.1 - - [27/Apr/2023 10:23:51] "GET /chain HTTP/1.1" 200 -
port is 5000
Pointer is at: 0
IT'S LOOP 1 RN
Pointer is at: 0
IT'S LOOP 2 RN
Pointer is at: 0
IT'S LOOP 3 RN
Pointer is at: 0
IT'S LOOP 4 RN
Pointer is at: 0
IT'S LOOP 5 RN
Pointer is at: 0
IT'S LOOP 6 RN
Pointer is at: 0
IT'S LOOP 7 RN
Pointer is at: 1
looping ... 2023-04-27 10:29:00.327846 ...
Successfully added tx from backend at 2023-04-27 10:29:02.370853
Out of loop
===== MINED A BLOCK SUCCESSFULLY =====
===== 2023-04-27 10:29:04.415961 =====
IT'S LOOP 8 RN
Pointer is at: 2
looping ... 2023-04-27 10:29:05.419783 ...
Successfully added tx from backend at 2023-04-27 10:29:07.471315
Out of loop
===== MINED A BLOCK SUCCESSFULLY =====
===== 2023-04-27 10:29:09.514957 =====
IT'S LOOP 9 RN
Pointer is at: 5
looping ... 2023-04-27 10:29:10.529786 ...
Successfully added tx from backend at 2023-04-27 10:29:12.580977
looping ... 2023-04-27 10:29:12.581976 ...
Successfully added tx from backend at 2023-04-27 10:29:14.620342
looping ... 2023-04-27 10:29:14.620342 ...
Successfully added tx from backend at 2023-04-27 10:29:16.662295
Out of loop
===== MINED A BLOCK SUCCESSFULLY =====
===== 2023-04-27 10:29:18.723056 =====
IT'S LOOP 10 RN
Pointer is at: 8
looping ... 2023-04-27 10:29:19.730206 ...
Successfully added tx from backend at 2023-04-27 10:29:21.773937
looping ... 2023-04-27 10:29:21.773937 ...
Successfully added tx from backend at 2023-04-27 10:29:23.823757
looping ... 2023-04-27 10:29:23.823757 ...
Successfully added tx from backend at 2023-04-27 10:29:25.875675
Out of loop
===== MINED A BLOCK SUCCESSFULLY =====
===== 2023-04-27 10:29:27.946140 =====
IT'S LOOP 11 RN
Pointer is at: 14
looping ... 2023-04-27 10:29:28.949755 ...
Successfully added tx from backend at 2023-04-27 10:29:30.995969
looping ... 2023-04-27 10:29:30.995969 ...
Successfully added tx from backend at 2023-04-27 10:29:33.032768
looping ... 2023-04-27 10:29:33.032768 ...
Successfully added tx from backend at 2023-04-27 10:29:35.079676
looping ... 2023-04-27 10:29:35.080677 ...
Successfully added tx from backend at 2023-04-27 10:29:37.121019
looping ... 2023-04-27 10:29:37.121019 ...
Successfully added tx from backend at 2023-04-27 10:29:39.185613
looping ... 2023-04-27 10:29:39.185613 ...
Successfully added tx from backend at 2023-04-27 10:29:41.255919
Out of loop
===== MINED A BLOCK SUCCESSFULLY =====
===== 2023-04-27 10:29:43.310149 =====
```

Figure28(1) simulation output

```

IT'S LOOP 12 RN
Pointer is at: 22
looping ... 2023-04-27 10:29:44.326735 ...
Successfully added tx from backend at 2023-04-27 10:29:46.373827
looping ... 2023-04-27 10:29:46.373827 ...
Successfully added tx from backend at 2023-04-27 10:29:48.417428
looping ... 2023-04-27 10:29:48.417428 ...
Successfully added tx from backend at 2023-04-27 10:29:50.462152
looping ... 2023-04-27 10:29:50.462152 ...
Successfully added tx from backend at 2023-04-27 10:29:52.506021
looping ... 2023-04-27 10:29:52.507021 ...
Successfully added tx from backend at 2023-04-27 10:29:54.558531
looping ... 2023-04-27 10:29:54.558531 ...
Successfully added tx from backend at 2023-04-27 10:29:56.612403
looping ... 2023-04-27 10:29:56.612403 ...
Successfully added tx from backend at 2023-04-27 10:29:58.679190
looping ... 2023-04-27 10:29:58.679190 ...
Successfully added tx from backend at 2023-04-27 10:30:00.739163
Out of loop
===== MINED A BLOCK SUCCESSFULLY =====
===== 2023-04-27 10:30:02.797594 =====
IT'S LOOP 13 RN
Pointer is at: 31
looping ... 2023-04-27 10:30:03.805898 ...
Successfully added tx from backend at 2023-04-27 10:30:05.853208
looping ... 2023-04-27 10:30:05.853208 ...
Successfully added tx from backend at 2023-04-27 10:30:07.895434
looping ... 2023-04-27 10:30:07.895434 ...
Successfully added tx from backend at 2023-04-27 10:30:09.943351
looping ... 2023-04-27 10:30:09.943351 ...
Successfully added tx from backend at 2023-04-27 10:30:12.001673
looping ... 2023-04-27 10:30:12.002674 ...
Successfully added tx from backend at 2023-04-27 10:30:14.050390
looping ... 2023-04-27 10:30:14.050390 ...
Successfully added tx from backend at 2023-04-27 10:30:16.104547
looping ... 2023-04-27 10:30:16.104547 ...
Successfully added tx from backend at 2023-04-27 10:30:18.165138
looping ... 2023-04-27 10:30:18.165138 ...
Successfully added tx from backend at 2023-04-27 10:30:20.216386
looping ... 2023-04-27 10:30:20.216386 ...
Successfully added tx from backend at 2023-04-27 10:30:22.266590
Out of loop
===== MINED A BLOCK SUCCESSFULLY =====
===== 2023-04-27 10:30:24.329064 =====
IT'S LOOP 14 RN
Pointer is at: 40
looping ... 2023-04-27 10:30:25.333436 ...
Successfully added tx from backend at 2023-04-27 10:30:27.382950
looping ... 2023-04-27 10:30:27.382950 ...
Successfully added tx from backend at 2023-04-27 10:30:29.423759
looping ... 2023-04-27 10:30:29.423759 ...
Successfully added tx from backend at 2023-04-27 10:30:31.465828
looping ... 2023-04-27 10:30:31.465828 ...
Successfully added tx from backend at 2023-04-27 10:30:33.506586
looping ... 2023-04-27 10:30:33.506586 ...
Successfully added tx from backend at 2023-04-27 10:30:35.548084
looping ... 2023-04-27 10:30:35.548084 ...
Successfully added tx from backend at 2023-04-27 10:30:37.602165
IT'S LOOP 17 RN
Pointer is at: 40
IT'S LOOP 18 RN
Pointer is at: 40
IT'S LOOP 19 RN
Pointer is at: 40
IT'S LOOP 20 RN
Pointer is at: 40

```

Figure28(2) simulation output

The time spent on cryptographic transactions and mining blocks is shown in the above figure. In total, the time taken to these procedures are both around 2 seconds. This can actually be expected, as both functions essentially use the same cryptographic algorithm.

The project has been calculated to generate around 800 bytes per transaction, which is 6400 bits. Counting with about 2 seconds per transaction/mining, the transfer rate would be around 3.2 kbps, which is far too slow for a commercially available web application. Therefore this project is only available as a demo and cannot be compared to a normal networked application.

The best way to improve the transfer rate is to find or write an efficient api for data transfer.

Another option is to use a multi-core processor and split the AES encryption algorithm into multiple modules on separate cores to achieve parallel computing, which can also be accelerated to some extent.

In terms of information transfer, it is really about reading and writing text files. This is functionally complete, but in practice it is risky. Unfortunately, no "c++ to solidity" tools were found. There is a library for "c++ to python", but given the speed of encryption in the blockchain system, it is still a difficult task to match the frequency of messages sent to the efficiency of messages received.

10. Conclusion

This project is an available logical result that I could submit in the time given. When I first chose the topic, I didn't really know anything about either blockchain or the Internet of Vehicles. After confirming my choice of topic, I read a lot of background literature and made time to learn the language I needed to use for development. At the beginning, I had high expectations for the project I was going to create. It should have a small but comprehensive IoV network emulation as the data output source, a corresponding api/abi for information transfer as a bridge between the two platforms, and the project should have used the appropriate content for the application scenario in terms of blockchain algorithms and consensus mechanisms. -- All this can also be found in the "Ideal design" of this article. However, during the actual coding process, I gradually realised that what I had planned could not be achieved with the available technology and time. So I have since developed a basic plan that is relatively simple in logic and basic in functionality, but with specific requirements that I am capable of implementing.

All in all, a logically sound result was presented, although there were errors in judgement of my technical ability. In addition, I gained considerable experience not only in blockchain architecture and algorithm research, but also in Solidity, C++, network simulation, usage of different frameworks, CMake and other small but important areas. Although these work can not be demonstrated in the project, I am satisfied.

11. Future Work

There are many areas where this project could be improved in the future.

The first is the wallet section, which should support multiple wallets in the database and allow users to browse and select specific wallets. Related back-end functionality should be added and the wallet class should have an id attribute for easy viewing and selection. The corresponding ui

should be updated and the user should be given a drop down list of all wallets recorded on the port. Wallets that are in active use should be highlighted.

The validation of input should be more detailed when transactions are sent and nodes are added. For example, for SQL injection attacks, we can detect the "<>" symbol in the background to reduce the possibility of javascript writes. For the data to be transferred, which is now just directly in the transaction object, a later improvement could be to pass in another special DataUpload class object as a parameter, and all data could be written in the DataUpload class.

The conflict fixing function should use a global variable (e.g. dictionary) to store the different versions of the chains and their respective current vote counts, and after looping all peer nodes, all chains that do not match the highest vote count will be overwritten. In addition, previous_hash can also be used to check for cheater if the chains are the same length but different in content; if a chain is judged to be cheating, the weight of that node will be reduced, (e.g. a node with a weight of 1 will now have a weight of 0.8).

For the network simulation part, the simple tictoc example is used to simulate the communication of nodes. It is not a vehicle simulation, but it does represent the essence of a vehicle network. Attempts were also made regarding the vehicle network emulation, but limited by the understanding of the c++ language, the extraction of information was not ultimately achieved. After greater understanding of the vehicle network emulation and designing a more suitable solution, it can be further emulated.

12. Reference

- [1]. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030 <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide>
- [2]. A.Hammoud, H. Sami, A. Mourad, H. Otrouk, R. Mizouni and J. Bentahar, "AI, Blockchain, and Vehicular Edge Computing for Smart and Secure IoV: Challenges and Directions," in IEEE Internet of Things Magazine, vol. 3, no. 2, pp. 68-73, June 2020, doi: 10.1109/IOTM.0001.1900109.
- [3]. K.Su, J. Li and H. Fu, "Smart city and the applications," 2011 International Conference on Electronics, Communications and Control (ICECC), 2011, pp. 1028-1031, doi: 10.1109/ICECC.2011.6066743.
- [4]. M. Cebe, E. Erdin, K. Akkaya, H. Aksu and S. Uluagac, "Block4Forensic: An Integrated Lightweight Blockchain Framework for Forensics Applications of Connected Vehicles," in IEEE Communications Magazine, vol. 56, no. 10, pp. 50-57, OCTOBER 2018, doi: 10.1109/MCOM.2018.1800137.
- [5]. S.Jiang, H. Fang and H. Wang, "Blockchain-Based Internet of Vehicles: Distributed Network Architecture and Performance Analysis," in IEEE Internet of Things Journal, vol. 6, no. 3, pp. 4640-4649, June 2019, doi: 10.1109/JIOT.2018.2874398.
- [6]. Annals of Emerging Technologies in Computing (AETiC), Print ISSN: 2516-0281, Online ISSN: 2516-029X, pp. 1-6, Vol. 2, No. 1, 1st January 2018, Available: <http://aetic.theiaer.org/archive/v2n1/p1.pdf>
- [7]. Z.Yang, K. Yang, L. Lei, K. Zheng and V. C. M. Leung, "Blockchain-Based Decentralized Trust Management in Vehicular Networks," in IEEE Internet of Things Journal, vol. 6, no. 2, pp. 1495-1505, April 2019, doi: 10.1109/JIOT.2018.2836144.
- [8]. Python Tutorial for Beginners - Learn Python by Building a Blockchain & Cryptocurrency <https://www.youtube.com/watch?v=KARxDX5DTgY&t=2140s>
- [9]. M.Magaia, Z. Sheng, P. R. Pereira and M. Correia, "REPSYS: A Robust and Distributed Incentive Scheme for Collaborative Caching and Dissemination in Content-Centric Cellular-Based Vehicular Delay-Tolerant Networks," in IEEE Wireless Communications, vol. 25, no. 3, pp. 65-71, JUNE 2018, doi: 10.1109/MWC.2018.1700284.
- [10]. K.M. Alam, M. Saini and A. E. Saddik, "Toward Social Internet of Vehicles: Concept, Architecture, and Applications," in IEEE Access, vol. 3, pp. 343-357, 2015, doi: 10.1109/ACCESS.2015.2416657.

- [11]. F.Ayaz, Z. Sheng, D. Tian and Y. L. Guan, "A Proof-of-Quality-Factor (PoQF)-Based Blockchain and Edge Computing for Vehicular Message Dissemination," in IEEE Internet of Things Journal, vol. 8, no. 4, pp. 2468-2482, 15 Feb.15, 2021, doi: 10.1109/JIOT.2020.3026731.
- [12]. Álvares, P.; Silva, L.; Magaia, N. Blockchain-Based Solutions for UAV-Assisted Connected Vehicle Networks in Smart Cities: A Review, Open Issues, and Future Perspectives. Telecom 2021, 2, 108-140. <https://doi.org/10.3390/telecom2010008>
- [13]. L.Kamal, G. Srivastava and M. Tariq, "Blockchain-Based Lightweight and Secured V2V Communication in the Internet of Vehicles," in IEEE Transactions on Intelligent Transportation Systems, vol. 22, no. 7, pp. 3997-4004, July 2021, doi: 10.1109/TITS.2020.3002462.
- [14]. Kapassa, E.; Themistocleous, M.; Christodoulou, K.; Iosif, E. Blockchain Application in Internet of Vehicles: Challenges, Contributions and Current Limitations. Future Internet 2021, 13, 313. <https://doi.org/10.3390/fi13120313>
- [15]. <https://medium.com/coinmonks/the-blockchain-473aac352e5>
- [16]. <https://oracle-patches.com/en/cloud-net/peer-to-peer-network-and-blockchain-technology>
- [17]. <https://phemex.com/academy/byzantine-fault-tolerance-bft>
- [18]. https://blog.csdn.net/weixin_38134491/article/details/85014562

13. Appendices

13.1 Log

28/09/2022

Initial discussion on the areas that should be covered by the project.

13/10/2022

Exchanged ideas and started working on the interim report

27/10/2022

Presentations were made on the current progress and feedback and suggestions were received. Further deepened the understanding of the topic and looked for a more suitable theme.

3/11/2022 - 17/11/2022

Confirm the topic and start practicing Solidity in remix Solidity IDE.

24/11/2022 - 8/12/2022

Exposure and learning the Truffle framework.

15/12/2022

Initially try to transfer data to the blockchain system constructed with Truffle framework.

22/12/2022 - 29/12/2022

No progress.

5/1/2023 - 26/1/2023

Develop python blockchain, complete parts: Block class; Blockchain class; Transaction class; Wallet class; utility. The system can be ran in console.

2/2/2023 - 23/2/2023

Try to dive into SUMO, Veins, C++, map extraction, etc.

2/3/2023 - 30/3/2023

Develop python blockchain, complete parts: access http method; add verification session; use TinyDB database; add Resolve Conflicts, File Checking function.

6/4/2023 - 20/4/2023

Write the dissertation.

13.2 Proposal Doc

Candidate number: 246743

Supervisor: Dr Naercio Magaia

Project Background

With the development of society and technology, the megatrend of the future is the Internet of Things. Everything is connected, allowing smoother and more unhindered communication between each other. One of the transport aspects of the Internet of Things is known as the Internet of Vehicles, which means that each vehicle can act as a node and be connected to a network through which efficient communication and information transfer can be achieved.

The applications of this technology are vast and promising, for example, using the Internet of Vehicles to build intelligent traffic systems, map the state of the road and give drivers road choices. Or the Social Internet of Vehicles, which allows passengers in vehicles in the same area to communicate with each other and allows drivers to exchange information in real time in case of possible vehicle breakdowns. In this sense, it could also make the driving environment safer.

But the fact is that because vehicles are highly connected, they will be more vulnerable to hacking

than ever before. Once a hacker takes control of the network, that can have serious and even costly consequences in terms of lives. This is where a storage method with a high level of security is needed to store data in the cloud. Blockchain is a good answer to this, as it is a distributed ledger that stores data in a robust form, and if a hacker were to try to hack into a traffic network, he would have to have very strong computing power or huge equity, which is almost impossible. Blockchain could therefore make vehicles more secure, transparent and auditable in the above scenario.

Aims & Objectives

Aims

My aim is to develop a blockchain-based system for storing information for connected cars, meaning I need to build the underlying code, which will use languages such as Python. Or some frameworks for Ethereum, as they are highly modular and can be implemented more easily for programming.

Objects

- Main Objectives

- Implement a blockchain-based data storage system that enables secure data storage, encryption, decryption and interoperability.
- Use appropriate consensus algorithms to improve efficiency while meeting security requirements.

- Extended Objectives

- To build a similar environment based on the characteristics of the connected car environment.
- To investigate whether there are alternative solutions for the subject I am focusing on.
- To check out the algorithms for smart contracts in detail.
- To design and simulate how to use a computer to simulate a real traffic environment.
- To design and arrange a complete set of testing requirements.

Relevance

The blockchain implementation will be written in python or ethereum. This is linked to the course I took in my sophomore year. This project will help me consolidate my knowledge of programming and broaden my horizons.

The choice of consensus algorithms and the implementation of several detailed techniques are also related to a module (Program Analysis) I have taken before.

Resources Required

A cloud server may be required to test the model.

Time available for the project

Mon	Tues	Wed	Thur	Fri	Weekends
12:00-15:00	16:00-19:00	11:00-15:00 OR 15:00-19:00	13:30-15:00	14:00-17:00	9:00-12:00 OR 14:00-17:00

Ps: May be adjusted according to actual workload.