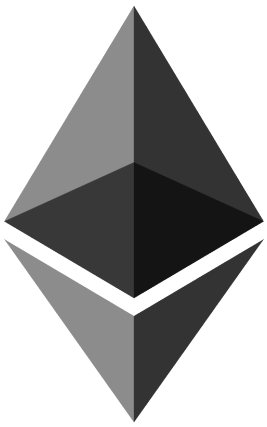
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



A Decentralized Application for Verifying a Matching Algorithm

Programming and Testing a Smart Contract on the Ethereum Blockchain

Master's thesis in Computer Science -Algorithms, Languages and Logic

LINNEA FRITZ

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

A Decentralized Application for Verifying a Matching Algorithm

Programming and Testing a Smart Contract on the Ethereum Blockchain

LINNEA FRITZ

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

A Decentralized Application for Verifying a Matching Algorithm
Programming and Testing a Smart Contract on the Ethereum Blockchain
LINNEA FRITZ

 Supervisor: Elad Schiller, Department of Computer Science and Engineering, Chalmers
Advisor: Henrik Fagrell, Diadrom
Examiner: Magnus Almgren, Department of Computer Science and Engineering, Chalmers

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

A Decentralized Application for Verifying a Matching Algorithm
Programming and Testing a Smart Contract on the Ethereum Blockchain
LINNEA FRITZ
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis uses blockchain technology to construct a decentralized application (often called a 'Đapp') for the sake of verifying results of a matching algorithm used on data in the automotive industry. Its main intent is to explore whether the framework Ethereum can be utilized to aid in ensuring the correctness of client responses to a query sent by a peer in the network. The application was programmed in Solidity and JavaScript, and run on a local test network consisting of five clients. Testing the finished application showed that the throughput of data was slow, approximately 35 bytes/s, and that taking over the network to send corrupted information was relatively simple. These findings, along with a general study of the areas where blockchain technology is most advantageous, led to the conclusion that though it has potential as a constituent in the car industry, it is not suitable for verification of matchings at the time of writing.

Keywords: blockchain, Ethereum, Bitcoin, Đapp, Truffle, P2P, matching algorithm.

Acknowledgements

I would like to thank my company advisor, Henrik Fagrell at Diadrom, my supervisor Elad Schiller, and my examiner, Magnus Almgren.

Linnea Fritz, Gothenburg, December 2017

Contents

List of Figures

List of Tables

1
Introduction

1.1 Context

Since the advent of computerized functions in cars, matching algorithms have also found a place in the automotive industry. Today, many of a car's features are realized by around 100 electronic control units (ECUs) communicating with each other via a network [7]. The term for such a computer is 'node', and each node has its own, unique identification number.

In order for a node to operate, it requires the correct combination of hardware and software. One node can be in production for many years, and accordingly, there will be many software and a number of hardware versions for it. For all nodes, there is at least one working software/hardware combination, and such a combination may apply to more than one node. Furthermore, there is a many-to-many relationship between hardware and software modules.

The fact that there are so many different available sets necessitates the use of databases to track and maintain the relations. That way, when a car comes to a repair shop, the mechanic is able to look up the node in question in a database and get an answer detailing which hardware fits at the node, and also which software module needs to be downloaded to run on this hardware.

To facilitate availability of this data to repair shops scattered across countries, each car manufacturer sets up their own servers, so that mechanics can send compatibility queries from different locations and at different times of day.

1.2 The issue with centralization

Though a common solution, centralized data repositories are prone to vulnerabilities such as distributed denial of service attacks [4] and compromised integrity. This entails, that the answers a client gets back may have been manipulated, compromised or gone altogether because of the nature of a client-server environment.

As a direct alternative to the centralized approach is the blockchain technology, which is inherently a decentralized and distributed solution. As such, its promoters testify that its traits of accountability and non-repudiation make it suitable for a wide range of applications such as storing documents, tracing provenience of prop-

erty and signing contracts [18]. The data is encrypted and can only be read by the holder of the private key.

We present a new solution for the problem of verifying hardware-software compatibility. To validate a hardware-software set found by a matching algorithm, we use the emerging blockchain technology, implemented on the Ethereum platform. This approach aims at providing increased scalability as well as improve service availability and user accessibility.

1.3 Problem statement

For a node to know which hardware/software composition it needs to run, an algorithm for database searches is required. Today, this setup is centralized, and all clients rely on a few central SQL servers to always be kept up to date, up and running, and ready to answer requests.

Reliance on central database servers leads to the dangers of single point of failure, where DDoS attacks, SQL injections and man-in-the-middle attacks are a constant threat to data integrity and availability. These vulnerabilities have prompted stakeholders to explore alternative solutions, and this is where blockchain technology comes in.

Compatibility checking in cars is best described as a matching problem, and in the sensitive environment of a car, verification of the combination's correctness is advisable. The problem for this thesis, then, is to design and implement a decentralized solution for clients to get verified answers to requests for finding compatible hardware/software sets.

The integrated development environment (IDE) proposed for this project, Truffle[1], is a framework specifically designed to facilitate development of decentralized applications. It uses Ethereum's blockchain implementation, and among other features, it allows "built-in smart contract compilation, linking, deployment and binary management, automated contract testing" and "network management for deploying to many public and private networks"[2].

As an Ethereum blockchain solution is commonly referred to as a Đapp, that is the term henceforth used in this thesis. The word is not just a contraction of "decentralized application"; the letter Đis pronounced "eth", which also refers to Ethereum's currency, ETH [20].

However, as Truffle is even more recent than Ethereum (its first version, 0.0.11, was released on GitHub in July of 2015[3]), it is still in its early stages, and there are

[1] `http://truffleframework.com/`
[2] `http://truffleframework.com/docs/`
[3] `https://github.com/ethereum/solidity/releases?after=v0.4.3`

multiple issues and bugs pending[4]. Though the ambition to make Truffle accessible for new users is clear in the documentation and tutorials[5], the limited amount of examples available online and the novelty of Đapps will make using the framework a challenge.

Not only are there bugs in Truffle, but its underlying structure, the Ethereum blockchain, has also faced a number of problems. These are described in recent error reports from Ethereum's official blog, and include a bug in the Geth client[6], storage corruption in the Solidity language[7] and a DDoS attack to the network[8]. Solidity has undergone 8 versions since the start of writing this thesis, and while most of the releases comprise of bugfixes, some come with new features as well[9]. This illustrates the fast evolution of the tools used when creating a Đapp, a fact that must be taken into consideration when planning a project using new technology.

Challenge: Designing and evaluating a decentralized blockchain application that validates the feasibility of an implementation of our matching algorithm (section 2.1). This challenge is advanced because using the studied platform and framework requires dealing with a number of code maturity issues, and poses difficulties in terms of adopting the available developmental tools.

1.4 Purpose

The main goal is to write a decentralized application for running compatibility checks for car nodes. Using blockchain technology, a network of computers will work together to process and answer calls from clients requesting information on which software versions can run on respective available hardware. The application will also include an error-reporting method, where users can log combinations that do not work, despite having gotten a positive response from the system. This may happen if there is a bug in the software module, or a software is wrongfully added to the database as backwards-compatible, when in reality, it is not.

As the existing solution is dependent on central servers, this thesis aims to explore the possibilities that decentralizing compatibility checks may hold. The pros and cons will be discovered through testing of the finished application, to disclose whether blockchain technology is a viable option for verifying the answer to a matching problem.

[4] https://github.com/consensys/truffle/issues

[5] http://truffleframework.com/tutorials/

[6] https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2/

[7] https://blog.ethereum.org/2016/11/01/security-alert-solidity-variables-can-overwritten-storage/

[8] https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/

[9] https://github.com/ethereum/solidity/releases

1.5 Thesis contribution

This dissertation provides crucial knowledge to the stakeholders (here, the automotive industry) interested in introducing a Blockchain application to distribute answers from a matching algorithm. In particular, we study whether a network of clients can work on a Blockchain to inform each other on which vehicular hardware and software modules are compatible, i.e., matching hardware and software components. Decentralizing this procedure has the potential of removing key vulnerabilities associated with central servers, such as man-in-the-middle and denial-of-service attacks. In order to be a viable option, the decentralized solution must be able to respond to matching queries quickly, do so on a large scale, and preferably perform at a low monetary cost.

We propose an implementation (based on Blockchain technology) of the matching algorithm by Edmonds [3] that is tailored to our industrial motivation for matching between hardware and software components. We use Blockchain technology as an enabler for providing a decentralized implementation. This thesis focuses on validating our implementation and checking whether it can meet all relevant requirements, such as the desired speed, throughput, and cost effectiveness.

Our test environment consists of five clients simultaneously running on one host, connected on a private Ethereum network. Each client runs its own instance of the decentralized application, which includes simulation of a local database. The tests are designed to measure the latency and throughput of the blockchain application, to see if it can compete with the centralized setup currently used. This is done using three different configurations (see chapter 4.5), where one client sends out over 2000 requests and each final answer received from the other clients is recorded to the blockchain.

The results obtained from the experiments show that the approximate data throughput is 36 bytes per second (see chapter 5.6), which leads us to draw the conclusion that Blockchain technology is too slow to compete with the current, centralized solution at this time, even taking into account security benefits that might come with decentralization. The tests also revealed that the Blockchain application has some issues when it comes to responding; at least twice during the test runs, it failed to reply to the calling client (see chapter 5.4). In one of these instances, the application, and all participating clients, needed to be restarted.

We offer to use this thesis for future reference as an example of a Blockchain application that is unsuitable for its intended purpose, underlining the importance of understanding when, how, and where a decentralized application is useful.

1.5.1 Test results

The result from testing the application is the main contribution of the project. Firstly, the results point to difficulties in designing and performing tests on the

blockchain. The time it takes to run a test in order to collect a minimum amount of data, coupled with flaws in the blockchain protocol, reveal the difficulty of testing the Ethereum blockchain on a desktop computer.

Secondly, the actual data from the tests validate the collusion theory; the blockchain will return incorrect answers if the network consists of a majority of selfish miners.

Combined, these findings sum up the core of the thesis. It is possible to design, program and test a decentralized application to use in the automotive industry. This thesis shows, however, that in this particular instance, there are few, if any, gains to migrating the matching procedure from a central to a distributed solution.

1.6 Scope

1.6.1 No Ethereum release

Though the real Ethereum blockchain is of interest to the subject, especially in terms of scalability and authentication (not everyone should be able to write data to this contract) the goal is not to deploy it officially. The reason for this limitation is partially a monetary question, it costs Ether (or, rather real money) to have a contract run on the blockchain. Another reason for not officially deploying the contract is that it greatly enhances risks of data leakage, and that, for this thesis, configuring a private testnet, and then running it on the official testnet, will suffice.

1.6.2 No cost assessment

As the contract will run only on the test networks, the "money" spent on it will not be real, and only Ether that is self-supplied (private testnet and RPC testnet) or easily mined, or worthless (mined and spent on Morden, the official testnet). The thesis is not specifically dealing with the currency aspect of blockchain technology, though in many cases, it is a vital part of it. Here, however, the main focus is assessing the usefulness of different clients reaching consensus on the answer to a matching function, and not the actual cost of the calculations.

1.6.3 No diagnostic tool interface

If there was a will to test the application in a more real setting, it would have been beneficial to integrate the Đapp with the diagnostic tool that is used today in the repair shops. However, in doing so, the focus of the thesis would move away from the blockchain and compatibility issues, toward a simpler programming task, which is best left to the future, should the need arise.

1.7 Method

The main method for researching and completing this thesis is a systematic review of literature on implementations of the blockchain protocol, studying existing decentralized applications and then investigating blockchain properties by programming on the Ethereum platform.

1.8 Thesis outline

In this first chapter, we outline the motivation and premises for conducting a study in blockchain technology, as well as offer some insight into the results and contribution of said study. Chapter two details the blockchain protocol, its strengths, weaknesses and applicability as a verifier of the matching algorithm. In chapter three, the thesis methodology is defined, and chapter four includes a description of the system and program code. Chapters five and six present detailed results from the tests performed on the code and system, and a discussion thereof, respectively. Lastly, chapter seven is a conclusion of the thesis findings.

2
Background

This chapter aims to give the reader a general introduction to the concept of blockchain technology. Firstly, section 2.1 offers a look at the current status of blockchains and recent usage of the technology that holds relevance to our thesis. In section 2.2, we outline the background of blockchain technology, its close relation to Bitcoin and a brief explanation of the protocol. Section 2.3 describes one of the most common challenges in cryptocurrency development: the double-spending problem, as well as one popular way of dealing with that issue. In section 2.4 we introduce Ethereum, the platform used to program and perform the tests of this project.

2.1 Blockchain technology today

Previously mostly known in general terms as the backbone of the cryptocurrency Bitcoin, the blockchain protocol has recently been acknowledged as a versatile technology. Under the title "IT trends", the journal IT Professional [13] states that the "potential range of blockchain utility" spans from the protection of intellectual property, to various applications on the Internet of Anything. The conviction is shared by Melanie Swan, author of the book "Blockchain –Blueprint for a New Economy" [18]. The book lists a decentralized domain name system, digital identity verification and decentralized governance services as future applications of the technology.

As this thesis is looking into the usage of blockchain technology in an automotive environment, Lee and Lee's [15] article on firmware updates is of particular interest. Instead of running their program on top of an existing blockchain implementation, they chose to write their own protocol. This alternative is outside of the scope of this thesis, but nevertheless an extension to keep in mind.

One representative decentralized application that has been implemented on a public blockchain is Betfunding [14]. It lets participants bet money on which crowdsourced projects will become reality. This can be seen as a typical use for blockchain, because it allows user participation, monetary transactions and a form of auction/voting process.

However, some researchers encourage careful consideration before jumping on the blockchain bandwagon. For example, Limieux's study on using Factom's[1] implementation of blockchain technology for "creating and preserving trustworthy digital

[1] `https://www.factom.com/`

records" [16] is not wholly optimistic. It concludes that, as it is designed now, blockchain may not be the way to go, much due to the fact that Factom does not offer data reliability.

Another downside to blockchain technology is the case of collusion between miners. Since a block is only accepted if a majority of miners agree on the solution to the problem (that is, they get a correct answer when using the same nonce as the miner who made the discovery first) it is fair to assume, that in order for a pool to gain control of Bitcoin's blockchain network, it needs to consist of $> 50\%$ of the participants.

However, Eyal and Sirer show that, if a minority of miners chose to follow a "selfish mining strategy", members of the honest majority will join them [10]. This process, modelled in a spatial Poisson process by Göbel et al [11], continues until the colluding miners sum up to more than half of participants, essentially centralizing the formerly decentralized currency. Such uses of the protocol can pose real dangers to uses of the blockchain, as it can break it by yielding incorrect computations.

2.2 Blockchain basics

2.2.1 Bitcoin

The most famous use of blockchain technology is Bitcoin [17], which was devised to allow trustless transactions between any participating members. The inventor(s), known only by the pen name Satoshi Nakamoto, wanted to introduce a payment method without the involvement of a third party, such as a bank or minter. With the tokens based on digital signatures (as is common), the new idea was to allow all nodes in the network to continuously calculate hash functions, to submit as proof of work. Recent figures suggest that Nakamoto was not alone in his pursuit of trustless monetary transactions, as the "total value of all existing Bitcoin currency equates roughly USD 7 billion" [1].

2.2.2 The protocol explained

Another term for blockchain is a "public ledger", a notion that literally describes the basic function of the technology. In a fully decentralized P2P network, all participants reposit the same copy of the blockchain, which is a structure storing all transactional data that has been sent between users since the very first block.

Each block consists of a block header, which, in turn, consists of a reference to the unique hash of the header of the block before it (32 bytes), a nonce (4 bytes), the difficulty target (4 bytes), the version (4 bytes), a timestamp (4 bytes) and the Merkle root of the transactions in the block [2]. For a look at the complete block headers for Bitcoin and Ethereum, please see appendices C.1 and C.2. The rest of the block is made up of the actual transactions that will be processed once the block has been mined.

The Merkle root is the hash of all transactions that are included in the block. If someone tries to make a change to a transaction, the hash of this transaction will no longer match the block's Merkle root, and by extension the hash of that block's header would change. This discrepancy alerts miners to an anomaly and they would not include that block in the chain.

The difficulty target is what a miner must reach when completing the proof-of-work, and in order to do this, she changes the value of the nonce in every iteration. The timestamp tells when the block was created, and the version number tracks upgrades to the software/protocol [2].

2.3 Double-spending, a common pitfall

However, the elimination of escrow services is not completely waterproof. A well-known problem when it comes to digital currency is the issue of double spending. When someone has paid for something with a token, what's there to prevent the same person spending the same token again? The common solution is the third party involvement, where, in the digital context, a minter will act as an intermediary.

Instead of paying the merchant directly, the customer gives his token to the minter, which issues another token of the same value and administers it to the receiver. Then all transactions must go via the minter. But in Bitcoin, there is no third party involvement, as it inherently is completely trustless, thus opening up for double-spending.

There are a number of double-spend attacks described in literature, such as the eclipse attack [5] and collusive mining [10]. In short, a double-spend attack on the blockchain will involve an attacker (A) (who may or may not be a participant on the blockchain), a receiving merchant (R) and the coin (C) that (A) supposedly pays with. (A) will send (C) as payment for goods to (R), and as soon as that transaction has been included in the latest block on the chain, (A) publishes two new blocks. In the first of these two blocks, she has included the same coin (C), but transfers it to another address she is in charge of. That way, the attacker now has both the goods she "purchased" from (R), as well as the money she originally spent [5].

Blockchain prevents double-spending through the proof of work consensus algorithm. In order for a participant to find a block ("mine") she must complete a very large number of calculations. She has succeeded when a calculation has yielded a "SHA-256 hash of the block's header that is lower than or equal to the current target"[2]. When a solution to the problem has been found, the finder broadcasts the solution to its peers, who in turn propagate the answer to all other nodes in the network. They all run the answer to verify its correctness, and when they find it to be true, they add that block to the chain. In so doing, the miner who found the node has

[2] `https://en.bitcoin.it/wiki/Target`

submitted proof of work - because the only way to find a hash is by working.

The attack described in above relies on the adversary being able to release two blocks at the same time, in order to provide a longer blockchain than the one that already exists (when the chain reaches a fork, the longest spoke always prevails). In Bitcoin, the recommended time to wait before actually considering a payment finalized, is six blocks. This reduces the attacker's chances of double spending, because mining six correct, consecutive blocks requires massive computational power; in October, 2017, the number of calculations in the Bitcoin network was between $6 * 10^{18}$ and $13 * 10^{18}$ hashes per second[3]. In his white paper , Nakamoto calculates the probability of an adversary mining six blocks before the rest of the honest network as P = 0.0002428. The probability becomes exponentially smaller with respect to number of blocks (z) generated after the transaction [17].

2.4 Ethereum

After Bitcoin, Ethereum is the second most popular blockchain based cryptocurrency [1]. Launched in July, 2015[4], it encompasses much more than just the ETH currency. It is a platform for developers to program so called smart contracts to be deployed to the blockchain. These smart contracts are written in the Turing complete language incorporated in the Ethereum blockchain protocols, and are, essentially, programs to be executed. That is, the transactions processed on the blockchain are not only monetary, but they can also be any type of data transfer between different applications, or between users.

For the code of a smart contract to be executed, the owner needs to pay a fee (called ether) that is accrued to the participant who mines the block with the code transaction in it. This 'gas' is proportional to the number and complexity of the calculations of the code execution.

[3] `https://blockchain.info/sv/charts/hash-rate`
[4] `http://ethdocs.org/en/latest/introduction/history-of-ethereum.html`

3
Method of accomplishment

Most documentation on Ethereum is written by members of the community, and can be found on sites such as stackexchange[1], reddit subforums[2] and Ethereum's own discussion boards[3]. Written material quickly becomes obsolete, as bug reports come in, are tended to and projects are abandoned or go through major changes.

Therefore, to find academic material, it is necessary to look to studies on blockchain technology, and Bitcoin papers. As Bitcoin has been around for a number of years and is a well-known cryptocurrency, it has attracted a lot of attention from scholars and programmers.

In addition to scouring the Internet and various journals for up-to-date information, the thesis relies on practical experiments. How these are prepared and implemented is described in sections 3.1 through 3.4. Finally, section 3.5 gives information on how the results of the experiments are to be evaluated.

3.1 Decentralized application design

In order to design the Đapp, it is important to understand what industry needs look like. To limit the scope of this thesis, the project will only consist of two methods to implement, but these have to present some sort of solution to a current problem, and need to be designed to fit what the users need and be somewhat integrated into the current system. This process will be done together with industry representatives who are familiar with the setup today. Even though there already is a solution in place, using central servers, this thesis will test a completely different approach.

Finding out the requirements for the Đapp means also finding out how it must perform and scale. These are also important parameters for the design process and making choices.

3.2 Programming

The Đapp will be programmed in Ethereum's programming language Solidity, which is reminiscent of JavaScript. For such a small application, with a limited number

[1] `https://ethereum.stackexchange.com/`
[2] `https://www.reddit.com/r/ethereum/`
[3] `https://forum.ethereum.org/`

of methods (two or three), no particular framework or IDE is needed. The Đapp is compiled by Solc which is the Solidity compiler that is built into Geth, which is Ethereum's client programmed in the Google language Go. This is necessary for bug fixing in the programming process.

3.3 Test design

Setting up a network to test the program in, means having to model a somewhat realistic situation, by using a constructed environment. Ethereum offers developers to design their own private testnets, where they configure a number of mock nodes (running on one, or more, machines), and these nodes need to know how to communicate with each other, which they do via ports.

To construct a suitable test environment for our experiments, a few alterations to Ethereum's default testnet need to be made. This includes deciding on the number of nodes in the network and setting them up with the passwords and addresses required even for faux transactions. Additionally, we have to program the Genesis block (see appendix B.3) to set the difficulty of the hashes to mine and the maximum size of a transaction, etc.

3.4 Mock data generation

In order to test database calls, and reproduce test results, a Java program to generate mock data into a spreadsheet could be an alternative to using an authentic SQL database. For the sake of simplicity, the data will be stored locally on each client that runs a blockchain node. Though there are P2P based databases available, learning how to use and implement them is not part of the scope. Future work can include trying a P2P solution, such as IPFS[4], STORJ[5] or Swarm[6]. This would still require a program to generate mock data, but it would be stored differently and require another type of integration solution for the blockchain.

3.5 Evaluation

The compatibility method will be evaluated in the testnet for two or three different traits, such as latency and block propagation. This will of course require the testnet to be configured for performing and measuring these evaluations.

Configuration of a private testnet in the Geth client allows the developer to choose how many nodes the network shall have (in this thesis, the private testnet will only include nodes that are running locally), and if this is successful, perhaps an extension to the tests would be to allow nodes running on other clients in the office to connect

[4] `https://ipfs.io/`
[5] `https://storj.io/`
[6] `http://swarm-gateways.net/bzz:/swarm/`

to this private network, to get a more realistic simulation, with more nodes involved.

The nodes will need to have access to the (in this case) mock database, and be able to communicate with each other. This is done by giving each node its own ports for listening and sending data.

4
System

In this chapter, we portray how the project's experiments are performed. We begin by detailing the technical specifics of our test environment in sections 4.1 and 4.2, and describe the process of creating and running our decentralized application in section 4.3.

We then move on to the evaluation part of our project. To highlight what parameters we look at in the test results, we define these key elements, *delay* and *throughput*, in section 4.4. In the following, and last, part of chapter four, we detail the structures of the reference test and two stress tests that the finished application undergoes.

4.1 Running the blockchain

For a user to participate on the blockchain, a client is required. As of February, 2017, users are offered seven different clients to use for Ethereum, among them Geth, written in Go, Parity in Rust, and Pyethapp in Python[1]. This thesis uses only one of the clients, Geth, because according to the Ethereum documentation, it is currently one of the leading implementations [9].

4.2 Geth

A client (called a node in the Ethereum community) can have none, one, or several accounts associated with it. This is similar to one person owning more than one bank account, perhaps one for savings, one for business, and another for private transactions. The client's purpose is communication with the blockchain; every time the client is started, it begins by synchronizing to the blockchain. This is in turn done by fetching information from the peers the node connects to upon startup.

The first time a node is instantiated, it needs to download the whole blockchain, at least if the purpose of running the node is to participate on the blockchain. Depending on the speed of its CPU/GPU and Internet connection, this may take from a couple of hours to days.

When the synchronization is complete, the node can start mining on the blockchain. In order to do this, it needs to be associated with an account in which to place the

[1] `http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html`

earned ether.

4.2.1 Nodejs and web3

Though an application can be run in the Geth client by using a JavaScript console and loading the Đapp's script, the functionalities of a client are restricted. Instead, in order to get full functionalities (such as imported nodejs modules) for the application, the Đapp's JavaScript code must contain an instantiation of web3, which is the JavaScript API used to integrate JSON RPC specifications with Ethereum applications. That way, the Geth client communicates with the Đapp via a web3 connection over RPC.

4.2.2 The private network

The Geth client runs the Ethereum protocol, which is then applied to the network that the user has specified. This can be the public, live Ethereum network (where ether is "real" money) the public live testnet Ropsten that allows users to mine ether quickly, so as to allow fueling tests, or a private network set up by the developer.

The private test network for this thesis consists of five clients all running on the same computer. To initiate a blockchain client, a script with the following commands[2] are run:

```
geth -unlock 0 -datadir="/path/to/ethereum/blockchain/node1"
-networkid 2017 -nodiscover -rpc -rpcport 8545 -rpcaddr 0.0.0.0
-verbosity 0 -rpccorsdomain "*" -port 30303 -rpcapi "net,personal,
eth,web3,admin" -ipcpath  /.ethereum/geth.ipc console
```

where each of the five clients modifies the parameters, such as the path to its directory.

Explanation:

`geth`: starts the client.

`unlock <value>`: as a client may have several accounts belonging to it, this command specifies the ID of the account to unlock for this session.

`datadir </path/to/nodedata>`: this is where information regarding the node in question is stored. The actual path is changed depending on node ID.

`networkid <value>`: the value of the network the user wants the client to connect to. Here, 2017 is a private network. For example, `networkid 1` means connection to the main, public Ethereum network.

`nodiscover`: disallows peers to find and connect to nodes on this private network (for security reasons).

[2] `https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options`

`rpc`: enables RPC for this node.

`rpcport`: which port this node will be listening for RPC calls on.

`rpcaddr <value>`: HTTP-RPC server listening interface (default value is: `localhost`)

`verbosity <value>`: how much information Geth will include in the log of the output file. 0 is nothing, while 6 is detailed.

`rpccorsdomain <value>`: this command takes a list of domains from which the client shall accept cross origin requests. In this command, the value `"*"` represents a wildcard, meaning that our browser will accept all requests to use the web page's resources, no matter the origin of the requesting domain.

`port <value>`: which port the network `2017` is listening on.

`rpcapi <value>`: APIs offered over the HTTP-RPC interface (default: `"eth, net,web3"`).

`ipcpath <value>`: filename for IPC socket/pipe within the data directory (explicit paths escape it).

Then, in order for all nodes on the network to do correct work (that is, reach consensus about the digits and protocol of the blockchain), they all need to be initiated to have the same seed block, known as the genesis block. Two nodes initiated from different genesis blocks cannot possibly work on the same blockchain; this is guaranteed by Ethereum's consensus algorithm.[3]. On the public blockchain, this file is pre-programmed (hard-coded), but to get more control over a private instance of it, the developer can write a bespoke genesis file. For an example, please see B.3.

4.3 Writing a program

As previously mentioned, the code needs to contain a web3 object, so that nodejs can communicate with the application. It also requires the setting of the http port for RPC, which is a parameter that needs to be passed for each node, because each node is running their instance of the Đapp, and as all nodes (in this thesis) are on the same computer, they need to listen to their own respective port. That means that the first node, which has RPC port 8545 will get that set as the in parameter for its app, the second node will listen to port 8546 and that is the parameter passed to the Đapp, and so on. That is the only parameter passed when starting the actual JavaScript part of the code.

[3] `http://ethdocs.org/en/latest/network/test-networks.html`

4.3.1 Truffle - a framework

The integrated development environment (IDE) proposed for this project, Truffle[4], is a framework specifically designed to facilitate development of decentralized applications. It uses Ethereum's blockchain implementation, and among other features, it allows the developer to compile, deploy and automatically test her smart contracts, as well as manage public and private network deployment[5].

However, as the Truffle framework is even more recent than Ethereum (its first version, 0.0.11 was released on GitHub in July of 2015[6]), it is still in its early stages, and there are multiple issues and bugs pending[7]. Though the ambition to make Truffle accessible for new users is clear in the documentation and tutorials[8], the limited amount of examples available online and the novelty of Đapps will make using the framework a challenge.

4.3.2 The code deconstructed

Now, let us consider the code running on a node. This code can be described, step by step, as follows:

1. A client sends out a request for a node identification number. What hardware/software set is compatible at this node?
2. On the private network 2017, all nodes are peers (know about and are connected to one another). This means that the call for information (data transaction) is broadcast to all clients on the network.
3. In order to prompt the other clients to also check the database, which consists of XLS tables (see `table1` and `table2` in figure 4.1), for an answer. When an original request is sent, the contract code fires an event that in turn triggers the JavaScript that the remaining clients are running.
4. So, the trigger calls the other clients to begin their search in the tables for the same node identifier and return an answer.
5. Every participating client contributes, by sending the match it got from the database to the Solidity contract.
6. The contract bubble sorts the answers (there are at most four answers at a time, which motivates the use of such a naive sorting algorithm) to make finding the most common answer easier.
7. When the sorting is done, the contract sends an event to a listener in the JavaScript code.
8. The JavaScript program does a "most popular" search on the returned array, which will then be the answer.

[4] `http://truffleframework.com/`
[5] `http://truffleframework.com/docs/`
[6] `https://github.com/ethereum/solidity/releases?after=v0.4.3`
[7] `https://github.com/consensys/truffle/issues`
[8] `http://truffleframework.com/tutorials/`

In the actual JavaScript application, a matching algorithm is run on the XLS files containing hardware and software data.

4.3.3 System architecture

In this section, we take a closer look at the inner workings of the system behind a single call for information on a `nodeId`. The illustration in 4.1 depicts the system constituents the call passes through.

1. `Script.sh` contains code to start the private network, compile the application, and serve it locally. It also generates a pseudo-random number to use for a compatibility check. The script opens the address of the web application and enters the node id `[0123]` in the query box.
2. The value `[0123]` is now saved in the web application's code as `nodeId`, and the query has triggered the `startCall` method in `App.js`.
3. `App.js`, in turn, calls the `startCall` method in the Solidity contract `App.sol` with `nodeId` as a parameter (still value `[0123]`).
4. The `startCall` method in `App.sol` will end only when all participating clients have completed a match search and the blockchain has reached consensus regarding the final answer.
5. To create a match to send to the blockchain, `App.js` calls its local database. It starts by looking up the node id, `[0123]` in `table1`.
6. The program then gets `[4567]` in return. This is the id of the hardware module that is compatible with node `[0123]`, and `App.js` saves it in the hardware variable.
7. `App.js` then calls `table2` with the hardware number as the parameter.
8. Finally, `table2` returns the id of a matching software module.

4.4 Definitions

In order to determine what the results from our experiments entail, we need to analyze two main outcomes of the tests:

- Delay. The time it takes to complete one transaction (here, a query). The start of the query is the moment a car node identifier is entered in the web application, and it finishes when the application returns an answer on the web page. Measurements have to be done in JavaScript, as there is not yet a feature for this in the Ethereum Geth client.
- Throughput. The number of transactions the Đapp can process per second. This measurement can be done by providing the `web3` command `getBlockTransactionCount` with the identification of the latest completed block. The output will be the amount of transactions within that block. This answer, divided by the time the block took to complete (calculated with timestamp) will provide the program's throughput.

4.5 Tests

The private test network consists of five Ethereum clients [$client_1, client_2, ..., client_5$] communicating via RPC. A request for an existing car node identifier is sent, the clients complete their calculations and the smart contract (Đapp) outputs the answer that most clients have returned. This follows the blockchain philosophy of 'what the majority answers, is deemed correct'.

Each node in the network takes on one of the following roles:

> c: caller. The client who sends out a transaction (query) on the blockchain.
> t: truthful clients responding to valid queries with correct data. For all t, where t= 0-100%, the first t clients always respond correctly to every query. If the query is invalid, that is, it regards a non-existent node, a truthful client will not answer anything.
> f: faulty clients, 100%-t. When active, they always return false data.

The tests do not utilize the matching algorithm, which, in either case, is implemented on the JavaScript side and not in the smart contract itself.

4.5.1 Reference test

A reference test is required in order to see that all parts of the application, the JavaScript program, the Solidity contract, the spreadsheet databases and the network are working as intended in a setting free from disturbances. The resulting data is then used for comparisons to the two other experiments, to discern what, if any, effects the introduction of faulty clients and invalid queries have on the outcome.

1. The script selects an *existing* node identifier for which to send a query. This may be done iteratively, beginning with node at row 0, and then incrementing until the last node in the database has been queried. It may also be done in a randomized fashion.
2. The JavaScript program is hard coded to always let $c = client_1$.
3. In the reference test, $t = 50\% + 1$, i.e., the majority. All t clients answer with a hard-coded {0, 0} tuple (combination), where the first value is the hardware, and the second is the software.
4. As the query regards an existing node with associated, correct data, all t clients will find a correct answer, and send this data to the blockchain.
5. There exist no f clients in this test, all clients respond with a correct answer.
6. On the blockchain, the smart contract finds that it has gotten 4 answers (from different clients), and triggers an event that the querying client, $client_1$ is listening for.

4.5.2 Stress test 1: valid queries

The first stress test we put our application through, is one where the querying client always sends out calls for an existent `nodeId`, but where some of the respond-

ing clients do not reply truthfully. The experiment is meant to mimic a common blockchain problem, where an adversary controls a number of participating clients and tries to gain advantage by falsifying transaction data.

Steps 1 and 2 are the same as in 4.5.1.

3. The first t clients behave predictably, as in the reference test. That is, they answer the calls with correct data.
4. The f clients, however, also respond in this test, but with falsified data. This means returning an answer that does not correlate to the node's database entry. Here, hard-coded to the value 1.
5. Every n tests, f is increased until the faulty clients reach a majority and can manipulate what the blockchain returns to the user. The plan is to increment f and decrement t by 1 (20%) every 200 queries, beginning at t=3 and f=1, until $f > t$.
6. The hypothesis is, that by getting f clients to collude and agree on sending the wrong answer, the Đapp will ultimately return an incorrect result to the user. In so doing, the test suite will showcase the Đapp's vulnerability, namely colluding clients.

4.5.3 Stress test 2: invalid queries

The motivation for this second stress test is to see what responses the blockchain return when the calling client requests data for a node id that does not exist in the database. In a blockchain with no adversaries, such a request poses no problem; if all participating clients are following the protocol, the caller will get a correct answer. However, we introduce faulty clients in this test in order to find out if their erroneous responses will be added to the blockchain as correct, if the percentage of faulty clients exceeds 50.

1. The script generates a *non-existent* node identifier for which to send a query.
2. The JavaScript program is hard coded to always let $c = client_1$.
3. The first t clients behave as predicted and return the value (-1), as the query is regarding a node that does not exist.
4. The faulty clients, however, generate a nonsense answer (value 1) and input it to the smart contract.
5. Every *200* tests, f is increased until the faulty clients reach a majority and can manipulate what the blockchain returns to the user. The plan is to increment f and decrement t by 1 (20%) every 200 tests, beginning at t=3 and f=1, until $f > t$.
6. When there are enough colluding f clients on the network, the Đapp will start returning faulty combinations to nodes that do not exist.
7. The hypothesis is the same as for 4.5.2.

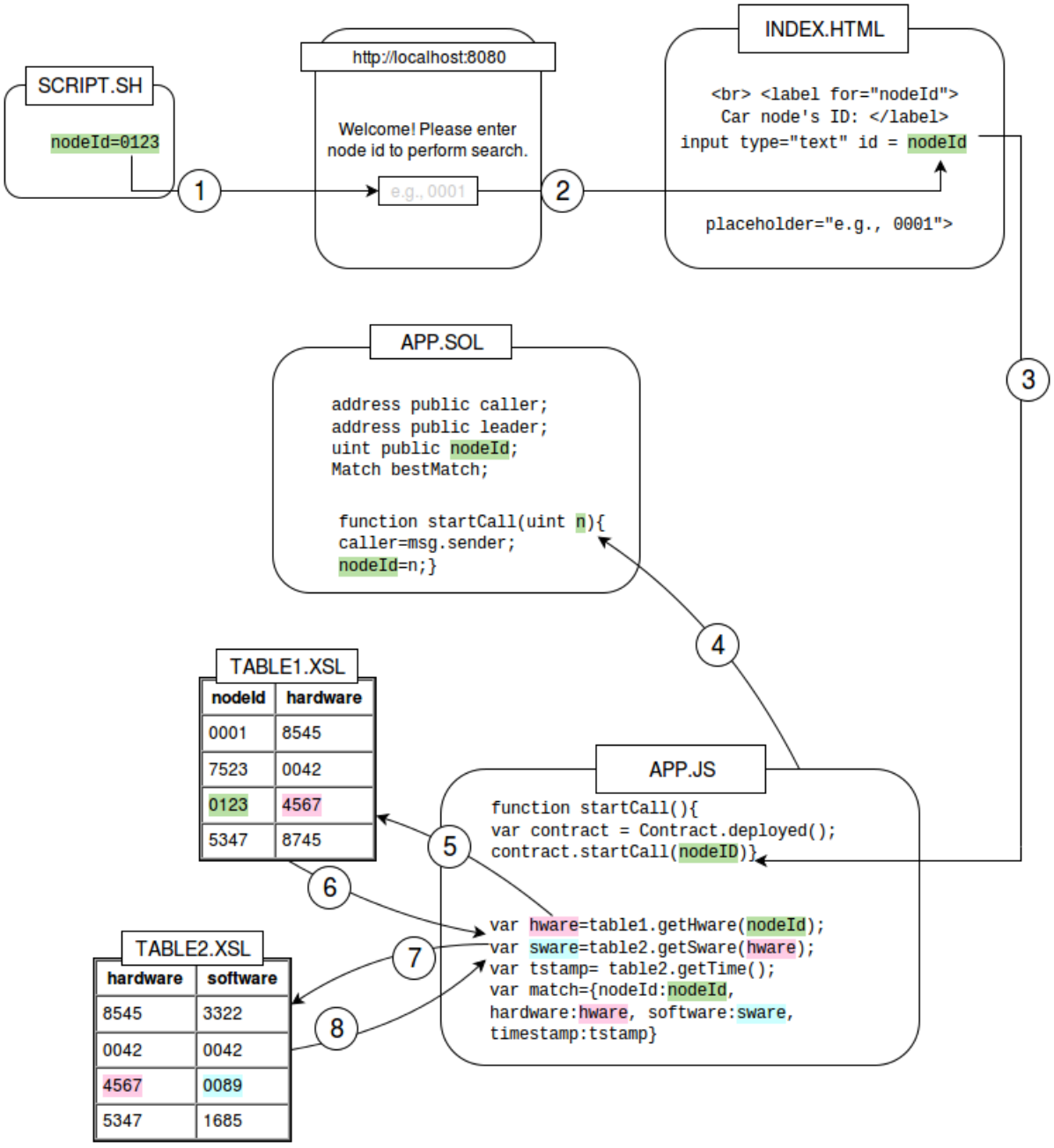


Figure 4.1: Illustration of the data flow when a query is entered in the web application.

5
Results

This chapter presents the findings of the three blockchain experiments run in the project. In section 5.1 and 5.2, scalability and throughput are discussed in terms of the reference test, which was performed with the intent of observing the general mechanisms of our distributed application. The data from the two stress tests are then discussed in one section each, 5.3 and 5.4, in relation to the reference test and this thesis' problem statement. Section 5.5 relates lessons learned from running the experiments, while the last part of this chapter is a summary of calculations performed on the test data, as well as tables depicting block generation on the blockchain.

In short, the results unambiguously point to inadequacies in a blockchain solution when it comes to efficiently replacing the existing, centralized systems. Using Ethereum's official protocol, the calculations take too long and require an untenable amount of CPU usage of the participating clients.

5.1 Latency in blockchains

There are two major components in a blockchain implementation that control the latency of transactions. One is the block size, and the other is the block frequency. Block frequency is how often a new block is mined (and this is dynamically established by changing the difficulty based on how long previous blocks took to process), for Bitcoin for example, this is approximately 12 minutes. Ethereum's goal time per block is 12 seconds[1], where in reality, it is closer to 19 seconds as of July, 2017 [2]. If previous blocks have taken less time to be added to the chain, this indicates that the difficulty is too low, and therefore, the next hash to crack becomes a little bit harder (a lower hash with more zeroes in the beginning) and the block frequency will slow down.

The other factor that influences the blockchain's latency is the block size, which determines how many transactions fit in a block. A large block means high throughput, because many transactions are processed in one go. However, increasing the throughput by increasing the block size would, by default, increase the total size of the whole chain (provided that there are many users sending transactions and taking advantage of the increased bandwidth) to enormous proportions [12]. That

[1] https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/
[2] https://etherscan.io/chart/blocktime

would in turn result in many fewer full nodes (which have the whole transaction ledger downloaded as well as relays and verifies transactions).

Moreover, computing thousands of transactions per second requires a powerful processor. A full node is not needed in order to mine (only the header is required to do that, not the whole ledger), but it is needed for storing the entire ledger. Fewer full nodes in the network would essentially centralize the blockchain, because it would mean a small number of users are the only ones to verify transactions, and that would mean risk of collusion (they can work together to verify erroneous transactions) and in essence remove the whole point of blockchain, which is decentralization.

Ethereum uses a middle road, so to speak, by not having the size of the block measured in data, but in ether - the sum of the cost for each transaction must not exceed the maximum, which is currently set to 4.7 million ether. This means, that if one very expensive transaction needs to be processed, it could entail the inclusion of only this single transaction in a block.

Figure 5.1 displays the result of our reference test, whose objective is to verify that our application behaves according to the blockchain protocol. The latency from our experiments closely follows the theoretical Ethereum blockchain protocol, with an average block time of 11.88 s. For a short summary of each test, see table 5.2.

5.2 Throughput in blockchains

As mentioned above, the throughput in Ethereum depends on the cost of the transactions to be included in a block. In our experiments, this results in a fairly stable throughput, as there are only two types of transactions, and they are performed at regular intervals. The two types of transactions are queries and responses, and each query costs 44,360 gas (ether), and each response costs 245,532 gas. In the real world Ethereum network, however, there are many types of transactions performed in each block, which means that the throughput constantly varies in terms of number of transactions per block, but not much in the maximum gas usage. The latter can, however, be adjusted up- or downwards by 1/1,024 per block by the miner.

In our experiments, as previously mentioned, a query costs 44,360 ether to run, and the cost limit for a block is 4,712,388 in the genesis file, which we configured (see appendix B.3) for the sake of performing the experiments. This means that, as $4,712,388/44,360$ equals 106, the theoretical amount of our particular queries could be 106 per block, which means 106/12 seconds $\approx 8,8$ transactions per second. However, as previously stated, our implementation does not allow for more than one query per block, and our test environment, as well as the application code, are configured to only accept 4 responses (or, rather, wait until the 4 responding accounts have all sent their result).

This means, that even though the calculation for the responses can be made in the same way as the calculation for the queries above, they cost 245532 to run, and with

total limit 4,712,388, we get a theoretical throughput of $4,712,388/245,532/12 \approx 1,6$ transaction (of 4 responses) per second. Thus, we are still limited by our test setup.

The limitations are set by the fact that the querying client, $client_1$, sends one request, and then waits until it has gotten the 4 responses before it sends a new request. And since the event for the request is not triggered in the responding clients until after the block has been mined, there is always the waiting time between sending a request, getting it included in a block that is then mined, and the clients getting triggered by the event firing.

5.3 Result of sending valid queries

Stress test 1, described in section 4.5.2 comprises of $client_1$ sending requests for an existing node. For the sake of experiments, the whole process is hard-coded; it is not the matching algorithm that is tested, but what responses are sent back to the request. As stated in the test description, the main objective was to "break" the blockchain, by getting it to start reporting incorrect responses as the answer to node requests. During the approximately 16 hour test run, the chain got stuck once, at block 11, and took 3,335 seconds to generate a new block. This can be observed in figure 5.2.

5.3.1 Queries 1-200

When sending out the first 200 queries, there is only one client, $client_5$, programmed to reply with an incorrect answer. The remaining 3 clients all answer correctly, and the querying client, $client_1$, gathers the answers and picks out the most popular answer (by way of the function `findBestCombination`). The data from these tests show 100% correct answers.

5.3.2 Queries 201-400

The output from the previously mentioned function `findBestCombination` will depend on the order of the input. If there is no "most popular combination", but rather a number of equally popular answers, the last input will be returned. At this stage in the test procedure, 50% of the returned answers were correct.

5.3.3 Queries 401 and up

As the majority of the responding clients have gone from t to f, the most common answer to all requests is 1, and that is the final response of the Đapp 100% of the time.

5.4 Result of sending invalid queries

Stress test 2 was performed in the same way as both the reference test and first stress test, with the difference being that $client_1$'s queries were regarding non-existent nodes. In theory, this means that the respondents would go through their respective database in search for the node in question, come up empty, and respond with the tuple {-1,-1}. A valid node matching must consist of a tuple of non-negative values, so -1 corresponds to "no answer", indicating a faulty query.

For unknown reasons, this test also froze, but for much longer than stress test 1, 12,425 seconds (3 hours, 27 minutes) at block number 88. Finally, we had to restart the test to get the chain working again (the error may have been caused by a block being mined so quickly that it was missed by the JavaScript listener), but this block skews the data. A plot including block 88 would not convey much information to the reader, so figure 5.3 depicts block 89 through 4824, omitting the erroneous block. In calculating the average block time, however, all blocks mined for stress test 2 are included.

5.4.1 Queries 1-200

Here, with only $client_5$ responding with incorrect data, the program returned the right answer, {-1,-1} every time.

5.4.2 Queries 201-400

Just as with the first stress test, when the number of corrupt clients rises to 20% of the total, half of the responses are correct.

5.4.3 Queries 401 and up

As soon as the number of colluding clients outnumber the genuine, the Đapp starts reporting incorrect answers 100% of the time.

5.5 Knowledge gained from the tests

As has been previously pointed out, one of blockchain technology's major weaknesses is the risk of colluding miners. The test result presented here are a variation of that - the blocks they return are correct, so they are not influencing the chain itself; however, the *data* they include in the blocks is incorrect. By way of synchronizing these incorrect answers, the f clients end up controlling not the chain, but the program residing on the chain.

On the plus side, as every transaction is inherently stored and timestamped, once a discrepancy is observed, tracing it is relatively easy. In stress test 2, for example, when the Đapp begins returning {1,1} combinations to non-existing nodes, the user will notice that something is wrong as soon as she applies that combination to the

Variable		Unit	Value
e_q	Cost for sending one query about one node.	Ether	$44,360$
e_r	Cost for sending the four required responses to one query.	Ether	$245,532$
s_q	The size of a query.	byte	681
s_r	The size of the four required responses to one query.	byte	$1,238$
g_l	Maximum amount of gas a block may cost. A block exceeding this amount will not be mined.	Ether	4,712,388
$start$	Starting time (in Unix time) of reference test.	second	$1,499,444,984$
fin	Finishing time (in Unix time) of reference test.	second	$1,499,506,639$
c_q	Number of queries sent in reference test.	-	$1,181$
c_r	Number of complete responses (four in each call) sent in reference test.	-	$1,079$
b	Time it takes to mine one block on Ethereum.	second	12

Table 5.1: Variables used for computing the calculations in section 5.6.

node. Using the address of her client and the approximate time her request was sent, one can filter out her call, see which addresses responded with a faux combination, and take appropriate action (such as ban these addresses from the blockchain).

5.6 Calculations

To better understand the proportions of data our Đapp is able to handle, we present calculations made to reach a number on both the theoretical and actual throughput of our blockchain application.

Disregarding the mechanisms of request/response messages, where a new request cannot be sent without having received a reply to the previous one, the potential number of requests in one block is:

$$g_l/e_q = 4712388/44360 = 106$$

And the potential number of complete responses (four for each request) in one block is then:

$$g_l/e_r = 4712388/245532 = 19$$

As the protocol is geared to add one block every 12 seconds, this gives a theoretical

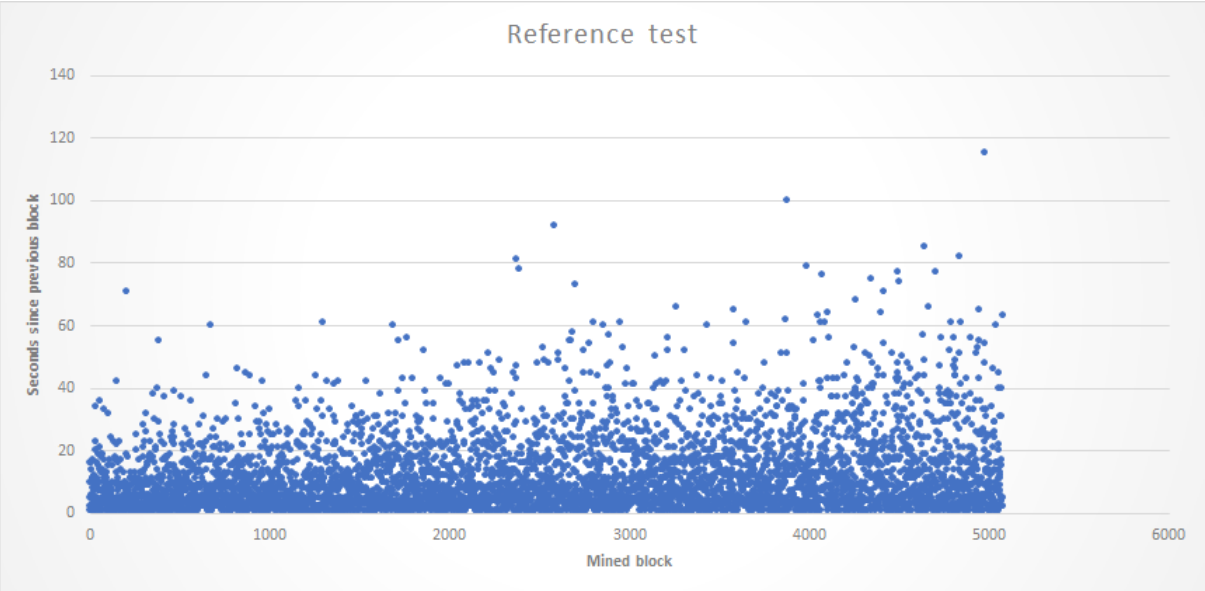


Figure 5.1: Plot depicting number of seconds required to mine one block in the reference test. Average time to mine one block in reference test: 11.88 s.

throughput of

$$((g_l/e_q) * s_q)/b = ((4712388/44360) * 681/12 = 6,015\ B/s$$

Whereas the throughput in our reference test is much smaller:

$c_q{*}s_q{+}c_r{*}s_r/(fin{-}start) = 1181{*}681{+}1079{*}1238/(1499506639{-}1499444984) = 35,85\ B/s$

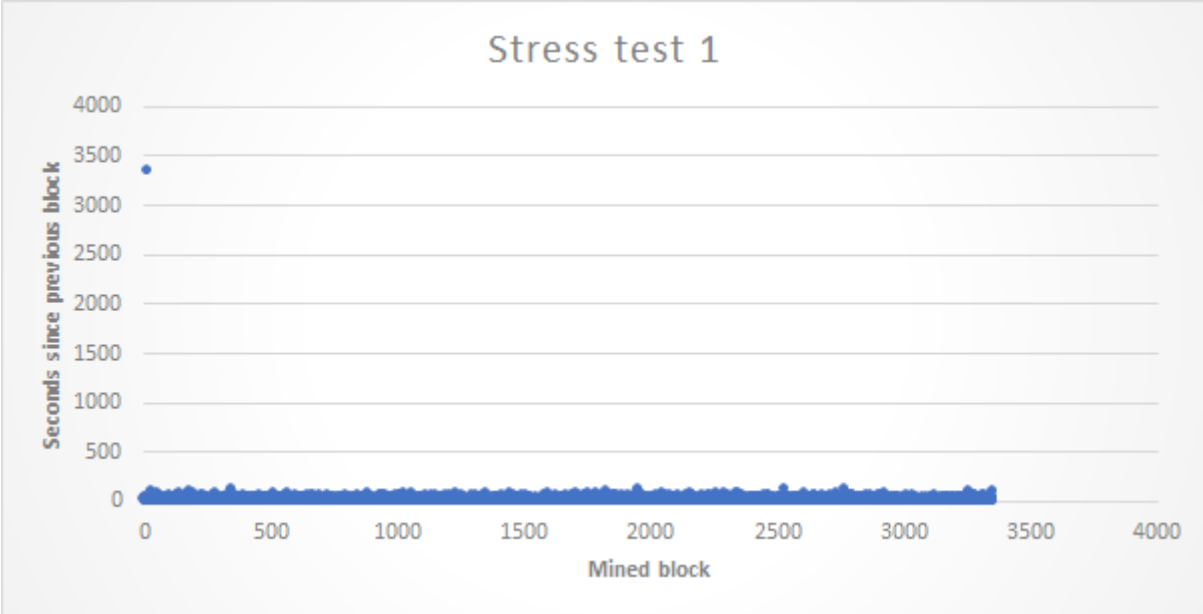


Figure 5.2: Plot depicting number of seconds required to mine one block in the first stress test. Average time to mine one block in stress test 1: 17.35 s. The outlier is explained in section 5.3.

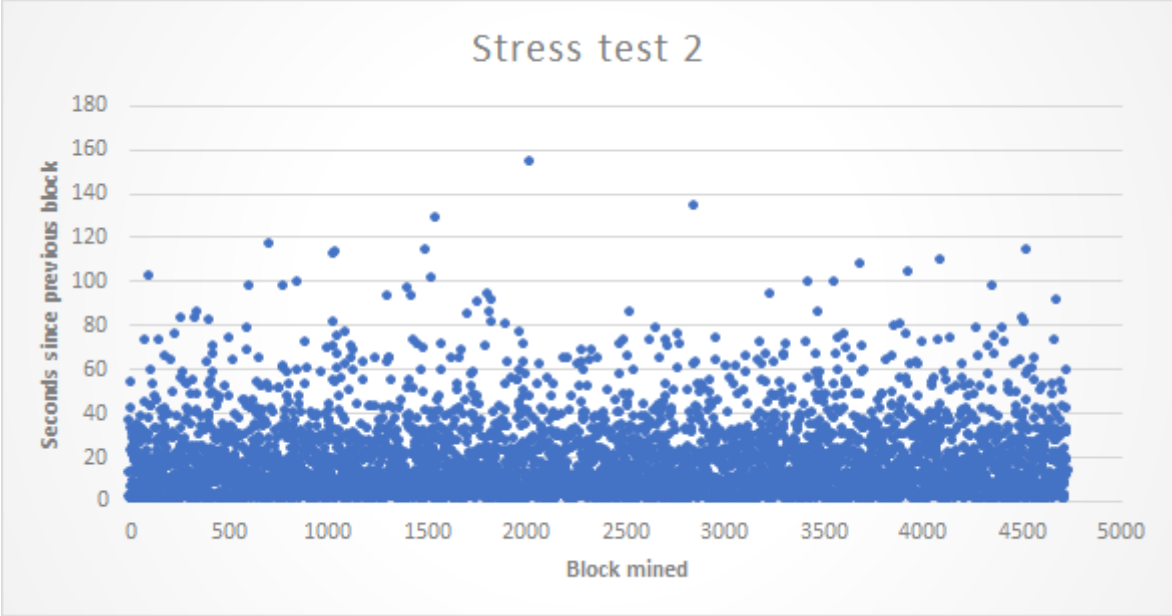


Figure 5.3: Plot depicting number of seconds required to mine one block in the second stress test. Average time to mine one block in stress test 2: 19.95 s.

Test	Objective	Invalid requests	Queries 0-200		Queries 201-400		Queries >400	
			c_t	c_f	c_t	c_f	c_t	c_f
Reference	Verify that the application adheres to specifications and that the blockchain on our private testnet is behaving as expected.	No	4	0	4	0	4	0
Stress test 1	Study the effects on the answers when faulty respondents are introduced on the blockchain.	No	3	1	2	2	1	3
Stress test 2	Study the effects on the answers when both faulty respondents and invalid requests are introduced on the blockchain.	Yes	3	1	2	2	1	3

Table 5.2: Overview of the three different tests run on the finished application.

6
Discussion

This part of the thesis is a reflection on both the theoretical and practical knowledge accumulated throughout the project. It only includes three sections. In the first portion, we reiterate the usages and premises of blockchain technology. In section 6.2, we reason about whether verification of a matching algorithm is a suitable problem to solve on the Ethereum blockchain, or not. Finally, section 6.3 includes a short summary of our thesis.

6.1 Deciding when a Đapp is the optimal solution

As stated in the beginning of this thesis, there are a number of areas where smart contracts are a very good idea. In their paper "Do You Need a Blockchain?", Wüst and Gervais list a number of questions that aid in the decision process. They can be summarized as follows [21]:

- Do a number of people need to write to the program?
- Can you use a database instead?
- Do you have a trusted third party to do verification instead?
- Do you need to store the state of the program?

In addition, it is important to think about the cost-effectiveness of running a smart contract on a blockchain. Is the cost of writing a smart contract (and, for usability also a JavaScript counterpart, as well as a GUI), deploying it, and then having users continuously accessing it worth the cost, either monetarily or powerwise, through mining? How does one assess whether it is worth the cost or not?

What type of program one is thinking about running on the blockchain also matters. The easiest type to transpose from centralized to decentralized is one that bears characteristics of a 'regular contract', where it is of utmost importance to save the state, i.e. store the signed contract on the blockchain. Even if the program contains heavy computations and/or database accessing, as long as these can be performed outside of the blockchain without compromising the program's integrity, there may be room for considering using a blockchain.

Then, there is the type of blockchain one needs, if the data has to be publicly verified (by unknown users on a permissionless blockchain such as Ethereum) or if it is enough to use a more private chain (permissioned), where trusting the verification

of fewer users, who are all known, suffices. In essence, it is advisable to be aware of the best features of blockchain, and carefully consider whether they are used in the program one wants to decentralize.

6.2 A Đapp for matching algorithms

So, is the subject of this thesis, getting the answer to a matching algorithm through a blockchain request, a good candidate for a smart contract? Considering the fact that the "problem" has already been solved, that is, there is already a setup for sending requests to a central database and getting (mostly) correct responses, the main objective of the thesis has been to explore whether blockchain would be an option for sending a request, letting clients look up the request in their respective databases, answer through the blockchain and then let the caller pick out the "best" answer.

Information gained from research, the implementation of the Đapp, and the test results point in a general direction. But first, let's summarize the facts below.

6.2.1 The Đapp's advantages

A decentralized application avoids one of the main problems with servers - which is relying on a single point of failure. In so doing, it safeguards against the common attacks against servers, such as DDos and man-in-the-middle attacks. One of the biggest advantages with blockchain technology is, of course, the fact that every state is saved.
In our example, what is stored in the state is:

- the caller's address and time of the request
- the respondent's address and response value
- the time of the responses and the time the final (4th) response comes in and an event is triggered at the caller's end

This data is useful when tracking whether all clients have the correct, updated database or not, if a caller gets a number of different answers to a request, even though they may all work (could be that some of the clients have an older version of the database) they should all be updated to contain the same information.

6.2.2 The Đapp's drawbacks

There are a few main reasons for not using Ethereum for solving our problem. The core problem, matching, does not require public verification per se. Assuming (which we do, as long as the opposite has not been proven) that the matching algorithm used is correct [8], it is the content of the database that needs checking. But the data cannot be publicly verified in our solution, because outside users on the blockchain do not have access to the data from the database (they are stored locally at each client). However, this does not mean that the contract cannot be deployed publicly, but it just means that one of the main features of blockchain is lost on our program.

6.2.2.1 Throughput

Though speed is not a core issue for the topic of this thesis, the time it takes from request to response needs to be taken into account. Would the waiting time constitute a hindrance for the calling party? If setting up a private Ethereum network, the process can be sped up by configuring low mining difficulty, allowing higher block frequency.

6.2.2.2 Rebuilding the existing system

The solution proposed in this thesis, with locally stored databases, would require a new system setup. At the moment, the databases are stored centrally on remote servers that the clients access. A solution where every client instead has its own database would entail both distribution and storage challenges. For a small network of five to ten clients, with a relatively small sized database as in the thesis, it's feasible. In the real world, thousands of clients with their own copy of a database poses maintenance difficulties, regardless of using smart contracts for keeping track of updates. However, the smart contract can be used, without any modifications, for clients accessing a central server instead. Then, the blockchain would be used as a confirmation of the responses' integrity.

6.2.2.3 Computational resource

One of the drawbacks of using blockchain technology is the sheer number of computations that are performed by each miner to find the right hash. This is neither cost-effective, nor environmentally sustainable, but as previously mentioned, may be justifiable if the transactions on the blockchain are deemed "valuable" enough to store.

6.2.2.4 Redundancy

The test network used for this thesis was permissionless. Even though the clients are aware of who their peers are, they are not configured to trust them based on that; their trust is based on the proof of work consensus algorithm.

In our case, getting verification from a much larger network would be unnecessary, we do not need thousands of responses to sort through and choose every time we send a query. This means that using a permissionless blockchain is actually redundant and we could make do with a Byzantine Fault Tolerance protocol which has higher performance and lower latency [19].

6.2.3 Alternative decentralized approaches

Running and evaluating the tests for this project has raised a number of questions to consider when considering a blockchain solution. Several articles in the literature review also presented different solutions and implementations that may be better suited alternatives, not only for matching software/hardware modules, but

other areas where the automotive industry could benefit from exploring blockchain technology. Two such examples are presented below.

6.2.3.1 IPFS as an alternative solution

Though the central theme of this thesis has been to investigate a matching algorithm, the major concern has not been whether or not the algorithm itself is correct, but rather, if the result of the matching is the same across several clients. As long as they have the same implementation of the algorithm (which they do) the only difference is the data upon which the algorithm is run. We have worked with locally stored databases, which means that there is a large risk they will differ, depending on updates.

There is currently more than one evolving solution utilizing the P2P qualities of blockchain technology by having clients store and maintain data across a network. For the purpose of storing medium to large data sets, as in the basis of this thesis, IPFS[1] is a promising alternative.

6.2.3.2 A customized blockchain

Another blockchain oriented way to distribute, update and check software for nodes in cars, is applying the ideas from Lee and Lee's study on firmware for embedded devices [15]. In their article, they design their own blockchain protocol, which allows for a more flexible implementation. Ethereum is meant to be an all-round platform for executing essentially any kind of smart contract. The differences between contracts used by a car manufacturer may not be very large, so constructing a bespoke blockchain protocol (that no one else can access) could be a beneficial approach.

6.3 Summary

This thesis is an investigation of the applicability of blockchain technology in a limited area of the automotive industry. It researches whether a common matching algorithm, closely related to bipartite graph search, can be successfully implemented on the Ethereum platform. By writing a program and putting it through different tests, our aspiration is to determine if our decentralized implementation would prove a workable alternative to today's solution with central servers.

The information these tests supply, coupled with documentation of the Ethereum platform and articles researching blockchain technology advantages and downsides, provides the basis for our discussion. Our project reveals that, despite the benefits of secure transmissions, tracking and saving of data transactions in a network, there are negative aspects of our implementation that cannot be ignored. This includes low throughput and ineffective use of computational resources. By reasoning about the implications of using our Đapp in a real-world setting, we hope to give a nuanced picture of the experiences gained in this project on blockchain technology.

[1] `https://ipfs.io`

7
Conclusion

Blockchain technology is still a new and exciting field, not only for computer scientists, but also from an economic, political and philosophical perspective. Completing this project has revealed the technology's theoretical uses and inner logic, as well as the concrete usability of the largest platform to date, Ethereum.

Through researching current projects and literature, writing code and setting up a test environment for a Đapp, we have reached a better understanding, not only of the practicalities, but also of the fundamental uses of blockchain technology.

It is clear, that the less code in the smart contract, the better. It is also evident, that the more delimited the smart contract, the better. And finally, it became progressively clear that the fewer calls made to the blockchain, the cheaper. Paired with the fact that a good Đapp fulfills its purpose of solving a problem through decentralization, from the lessons learned in this thesis follows that blockchain technology is not the ultimate answer to our problem. Using blockchain technology to automatically verify matchings is simply too inefficient to use on a large scale.

That said, there are several other areas within the automotive industry where blockchain technology is worth exploring, such as tracking repairs and saving ownership contracts.

A

Detailed problem definition

Using common notation to model the problem, for each car, there is a set of nodes $N = n_1, ..., n_i$, a set of hardware modules $H = h_1, ..., h_k$ and a set of software modules $S = s_1, ..., s_j$. For every n, there is at least one combination $(h, [s])$ where hardware maps to a list of software it is compatible with.

Assuming the node id will work as a key in the database hash table, the worst time complexity is $O(n)$. The value in the hash table will be a list of hardware that can be used at that node. Here, the algorithm can pick e.g. the last element and use that as a new key. The next step is to use that key to search another hash table, to find software that can run on that hardware module. Again, the time complexity is, at worst, $O(m)$. The actual complexity will depend on the real implementation of the search, but it will not be greater than $O(mn)$ [8].

The problem of matching hardware and software closely resembles a unweighted bipartite graph problem, assuming that no heuristics are introduced. Then, graph G is equal to (H, S, E), with E being the set of edges, where an edge means that (h, s) are compatible. Finding the maximum number of matches, where no two edges share an end-point, can be resolved in $O(\sqrt{n}m)$, using Edmonds' algorithm [3]. As the base case here is just concerned with finding *any* matching s to an h, the complexity will be lower, since no more edges need traversal as soon as a match is found. Thus, the aim is to find a solution that does not result in complexity worse than polynomial time.

There are two possible extensions to the problem. One is to introduce heuristics to the combinations. For example, it is more expensive (but in the long run, perhaps more beneficial) to replace old, still functioning, hardware with the newest combination. Or, it could be cheaper to use the newest package because it needs to be tried in the aftermarket for evaluation.

The second suggested extension is to introduce a voting mechanism for users. This would enable the application to make use of one of blockchain's main features, namely, reaching consensus through participation. In such an extension, a user may introduce a change to the database, saying that node X can run (Y,Z) but not (Y,W), as the database states. The user has tried the combination and found it is erroneous, and therefore should be changed. This proposed change is entered in the application, sent to the blockchain to prompt users to try the combination themselves. If others find the combination faulty, too, they can vote yes, otherwise

no. This extension will require analysis of strategic voting, how to avoid it, as well as coming to a conclusion on how to decide when a vote is final (some nodes have higher status/veto? or just a percentage? etc).

B

Code

B.1 Javascript code

```
const Web3 = require("/usr/lib/node_modules/web3");
const web3 = new Web3();
const fs = require('fs');
const readline = require('readline');
const ports=["http://localhost:8545",
"http://localhost:8546",
"http://localhost:8547",
"http://localhost:8548",
"http://localhost:8549",
"http://localhost:8550",
"http://localhost:8551",
"http://localhost:8552",
"http://localhost:8553",
"http://localhost:8554"];


var contractAddress = fs.readFileSync('/home/linnea/test1/
   test1_addressInfo.txt','utf8');
var contractABI = fs.readFileSync('/home/linnea/test1/
   test1_abi.txt','utf8');
var contract = web3.eth.contract(JSON.parse(contractABI));
var myContract = contract.at(contractAddress);
var workbook;
var sheet_name_list;
var node_exists = true;
var counter =0;
var args = process.argv.slice(2);
var currentBlock1 =0;
var currentBlock2 =0;



module.exports = function(callback) {};
createWorkBook();
```

```
setProv(args[0]);
//test(web3.eth.blockNumber);
startProgram(args[0]);
//printEvents();
function setProv(n){
  web3.setProvider(new web3.providers.HttpProvider(ports[n
     -1]));
}

function startProgram(clientNumber){
  if (clientNumber==1) {
    process1(0);
  }
  else {
    process2();
  }
}

function process1(k){
  myContract.query(k, 2000000, {from: web3.eth.accounts[0]})
     ;
  var responding = myContract.responding({}, {fromBlock: '
     latest', toBlock: 'latest'});
  responding.watch((error, eventResult) => {
    if (error)
    console.log('Error: ' + error);
    else{
      console.log('myEvent: ' + JSON.stringify(eventResult))
         ;
      k = eventResult.args.r;
      var answers=[];
      console.log("\n Current block: " + currentBlock1 + "
         blocknumber: " + eventResult.blockNumber + '\n');
      if(currentBlock1!=eventResult.blockNumber){

        currentBlock1=eventResult.blockNumber;
        k++;

        for(var i=0; i<3; i++){
        var res=myContract.getMatch.call(i, 2000000,{from:
           web3.eth.accounts[0]});
        answers[i]=parseInt(res);
        console.log("answer from call: " + res);
        //console.log("i= " +res );

      }
```

```
      findBestCombination();
      myContract.query(k, 2000000, {from: web3.eth.accounts
         [0]});
    }
    else {
      console.log("same block number!");
    }
    }
  });
}


function process2(){
  var asking = myContract.asking({}, {fromBlock: 'latest',
     toBlock: 'latest'});
  asking.watch((error, eventResult) => {
    if (error)
    console.log('Error: ' + error);
    else{
      console.log('myEvent: ' + JSON.stringify(eventResult)
         + '\n');
      k = eventResult.args.q;
      console.log("\n Current block: " + currentBlock2 + "
         blocknumber: " + eventResult.blockNumber + '\n');
      if(currentBlock2!=eventResult.blockNumber){
        currentBlock2=eventResult.blockNumber;
        k=Math.floor(Math.random() * (10 - 1 + 1)) + 1;
                  //k++;
          myContract.response(k, 2000000, {from: web3.eth.
             accounts[0]});
        }
        else {
          console.log("same block number!");
        }
    }
  });
}

function createWorkBook(){
  if (typeof require !== 'undefined') XLSX = require('/usr/
     lib/node_modules/xlsx');
  workbook = XLSX.readFile('/home/linnea/callback/out.ods');
  sheet_name_list = workbook.SheetNames;
}

function table(id, sheet_number){
```

```
  var worksheet = workbook.Sheets[sheet_name_list[
     sheet_number]];
  var range = worksheet['!ref'];
  var myRange = XLSX.utils.decode_range(range);
  var end = myRange.e.r;
  for(var i=1; i<=end; i++){
    var row = 'A'+i;
    var value = worksheet[row].v;
    if(value == id){
      console.log("value: " + value);
      return (worksheet['B'+i].v);
    }
  }
  return 0;
}

function matchingAlgo(number){
  var answer1 = table(number, 0);
  console.log("answer1:" + answer1);
  var answer2=0;

  if (answer1!=0){
    answer2=table(answer1, 1);
  }

  if (answer2!=0){
    console.log("answer 1: " + answer1 + " and answer 2: " +
        answer2);
    node_exists=true;
    var answers = [answer1, answer2];
    return answers;
  }
  else console.log("that node does not exist!");
};


function findBestCombination(answers)
{
  var count = 1;
  var tempCount;
  var bestCombo = answers[0];
  var tempCombo = 0;

  for (var i = 0; i < answers.length ; i++)
  {
    tempCombo = answers[i];
```

```
    tempCount = 0;
    for (var j = 1; j < answers.length; j++)
    {
      if (tempCombo.hardware == answers[j].hardware)
      tempCount++;
    }
    if (tempCount > count)
    {
      bestCombo=tempCombo;
      count = tempCount;
    }
  }
  console.log("bestCombo: " + bestCombo.hardware + ", " +
     bestCombo.software + ", " + bestCombo.address + 'n');

}
```

B.2 The Solidity smart contract code

```
pragma solidity ^0.4.9;

contract Match{

  function Match() payable {
  }

  struct Comp{
    int hardware;
    int software;
  }

  Comp[] matches;
  event asking (address sender, uint q);
  event responding ();

  function query(uint q)
  {
    delete matches;
    asking(msg.sender, q);
  }

  function response(int h, int s)
  {
    matches.push(Comp(h, s));
    if (matches.length>3){
      bubbleSort();
```

```
    }
  }

  function bubbleSort()
  {
    uint n = matches.length;
    while (n>0){
      uint v = 0;
      for (uint i=1; i<= (n-1); i++){
        if (matches[i-1].software>matches[i].software)
        {
          swap(i-1, i);
          v=i;
        }
      }
      n=v;
    }
    responding();
  }

  function swap(uint index1, uint index2)
  {
    Comp memory temp = matches[index1];
    matches[index1] = matches[index2];
    matches[index2] = temp;
  }

  function getMatch(int m) returns (int, int)
  {
    return (matches[m].hardware, matches[m].software);
  }
}
```

B.3 The genesis file

```
{
"nonce": "0x0000000000000042",
 "timestamp": "0x0",
 "parentHash": "0x0000000000000000000000000000000000000000",
 "extraData": "0x0",
 "gasLimit": "0x800000",
 "difficulty": "0x400",
 "mixhash": "0x0000000000000000000000000000000000000000",
 "coinbase": "0x3333333333333333333333333333333333333333"
 }
```

C

Blockchain resources

C.1 Bitcoin block header

The following information is an excerpt from the Bitcoin Developer Reference [6].

version The block version number indicates which set of block validation rules to follow.

previous block header hash A SHA256 hash in internal byte order of the previous block's header. This ensures no previous block can be changed without also changing this block's header.

merkle root hash A SHA256 hash in internal byte order. The merkle root is derived from the hashes of all transactions included in this block, ensuring that none of those transactions can be modified without modifying the header. See the merkle trees section below.

time The block time is a Unix epoch time when the miner started hashing the header (according to the miner). Must be strictly greater than the median time of the previous 11 blocks. Full nodes will not accept blocks with headers more than two hours in the future according to their clock.

nBits An encoded version of the target threshold this block's header hash must be less than or equal to.

nonce An arbitrary number miners change to modify the header hash in order to produce a hash less than or equal to the target threshold. If all 32-bit values are tested, the time can be updated or the coinbase transaction can be changed and the merkle root updated.

C.2 Ethereum block header

The following information is an excerpt from the Ethereum yellow paper [20].

parentHash The Keccak 256-bit hash of the parent block's header, in its entirety; formally H_p.

ommersHash The Keccak 256-bit hash of the ommers list portion of this block; formally Ho. beneficiary: The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally H_c.

stateRoot The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied; formally H_r.

transactionsRoot The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; formally H_t.

receiptsRoot The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; formally H_e.

logsBloom The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list; formally H_b.

difficulty A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; formally H_d.

number A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero; formally H_i.

gasLimit A scalar value equal to the current limit of gas expenditure per block; formally H_l.

gasUsed A scalar value equal to the total gas used in transactions in this block; formally H_g.

timestamp A scalar value equal to the reasonable output of Unix's time() at this block's inception; formally H_s.

extraData An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; formally H_x.

mixHash A 256-bit hash which proves combined with the nonce that a sufficient amount of computation has been carried out on this block; formally H_m.

nonce A 64-bit hash which proves combined with the mix-hash that a sufficient amount of computation has been carried out on this block; formally H_n.

Bibliography

[1] Luke Anderson, Ralph Holz, Alexander Ponomarev, Paul Rimba, and Ingo Weber. *New Kids on the Block: an Analysis of Modern Blockchains*. In: *arXiv preprint arXiv:1606.06530* (2016).

[2] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, Inc., 2014.

[3] Jørgen Bang-Jensen and Gergory Gutin. *Digraphs: Theory, Algorithms and Applications*. 1st ed. London: Springer, 2002.

[4] Udi Ben-Porat, Anat Bremler-Barr, and Hanoch Levy. *Evaluating the Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks*. In: *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE. 2008, pp. 2297–2305.

[5] George Bissas, Brian Neil Levine, A Pinar Ozisik, Gavin Andresen, and Amir Houmansadr. *An Analysis of Attacks on Blockchain Consensus*. In: *arXiv preprint arXiv:1610.07985* (2016).

[6] *Bitcoin Developer Reference*. Website. [Online; accessed 25-Jul-2017]. URL: `https://bitcoin.org/en/developer-reference`.

[7] Robert M Charette. *This Car Runs on Code*. Blog post. [Online; accessed 25-11-2017]. 2009. URL: `https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code`.

[8] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. Massachusetts Institute of Technology, 2009.

[9] *Ethereum Homestead Documentation*. [Online; accessed 2017-11-19]. URL: `http://ethdocs.org/en/latest/`.

[10] Ittay Eyal and Emin Gün Sirer. *Majority Is Not Enough: Bitcoin Mining Is Vulnerable*. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2014, pp. 436–454.

[11] Johannes Göbel, Paul Keeler, Anthony E Krzesinski, and Peter G Taylor. *Bitcoin Blockchain Dynamics: the Selfish-Mine Strategy in the Presence of Propagation Delay*. In: *Performance evaluation* 104 (2016), pp. 23–41.

[12] *Hard Problems of Cryptocurrency*. Wiki. [Online; accessed 25-07-2017]. 2016. URL: `https://github.com/ethereum/wiki/wiki/Problems/`.

[13] George Hurlburt. *Might the Blockchain Outlive Bitcoin?* In: *IT Professional* 18.2 (2016), pp. 12–16.

[14] Viktor Jacynycz, Adrian Calvo, Samer Hassan, and Antonio A Sánchez-Ruiz. *Betfunding: A Distributed Bounty-Based Crowdfunding Platform over Ethereum*. In: *Distributed Computing and Artificial Intelligence, 13th International Conference*. Springer. 2016, pp. 403–411.

[15] Boohyung Lee and Jong-Hyouk Lee. *Blockchain-based Secure Firmware Update for Embedded Devices in an Internet of Things environment*. In: *The Journal of Supercomputing* (2016).

[16] Victoria Louise Lemieux and Elizabeth Lomas. *Trusting Records: Is Blockchain Technology the Answer?* In: *Records Management Journal* 26.2 (2016).

[17] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Tech. rep. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[18] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 2015.

[19] Marko Vukolić. *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*. In: *Open Problems in Network Security: IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*. Ed. by Jan Camenisch and Doğan Kesdoğan. Springer International Publishing, 2016, pp. 112–125. URL: https://doi.org/10.1007/978-3-319-39028-4_9.

[20] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Tech. rep. 2014. URL: http://gavwood.com/paper.pdf.

[21] Karl Wüst and Arthur Gervais. *Do You Need a Blockchain?* In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 375. URL: https://eprint.iacr.org/2017/375.pdf.