CHALMERS
UNIVERSITY OF TECHNOLOGY

Decentralized Cloud Computing Platforms

Building a Global Marketplace for Computing Power

Bachelor of Science Thesis in Computer Science

JOHAN ANGSÉUS
ADRIAN BJUGÅRD
WILHELM HEDMAN
JOHAN LINDSKOGEN
OSKAR NYBERG
JOEL TORSTENSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Bachelor of Science Thesis

Decentralized Cloud Computing Platforms

Building a Global Marketplace for Computing Power

JOHAN ANGSÉUS
ADRIAN BJUGÅRD
WILHELM HEDMAN
JOHAN LINDSKOGEN
OSKAR NYBERG
JOEL TORSTENSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2015

Decentralized Cloud Computing Platforms
Building a Global Marketplace for Computing Power
JOHAN ANGSÉUS
ADRIAN BJUGÅRD
WILHELM HEDMAN
JOHAN LINDSKOGEN
OSKAR NYBERG
JOEL TORSTENSSON

Supervisor: Lennart Hansson, Department of Computer Science and Engineering
Examiner: Arne Linde, Department of Computer Science and Engineering

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

Cover illustration: Cloud anchor, by the authors.

Gothenburg, Sweden 2015

Abstract

This thesis describes a basis for decentralized cloud computing platforms; a suggested protocol to be used for such a platform; and places decentralized networks and their origins in a technical and practical context. The problem domain includes how to distribute work and how to verify computations within a decentralized network. To mitigate attacks on the network by internal and external actors, a combination of incentives and free market principles are used. The protocol is utilizing a general-purpose blockchain as well as other more established forms of network communication. Arbitration between network participants is realized using smart contracts.

A reference implementation of the protocol, named *Zeppelin*, has been partially realized in the Ethereum general-purpose blockchain, and deployed on a small-scale network. The reference implementation is highly modular and demonstrates the ability for decentralized applications to use both a traditional backend and a blockchain-based backend. By using a blockchain, some application data and business logic is stored and executed on a global virtual machine, distributed between participating nodes. The reference implementation should be regarded as a proof-of-concept of the proposed protocol, and is not yet ready for a production release. This is largely attributed to the fact that general-purpose blockchains are currently in a very early development phase and can not yet be used reliably.

Keywords: decentralization, blockchain, trustless, consensus, Ethereum, smart, contract, Solidity.

Sammanfattning

Denna rapport sätter decentraliserade nätverk i ett tekniskt och praktiskt sammanhang, samt beskriver en grund för decentraliserade molnbaserade beräkningsplattformar genom att föreslå ett protokoll som kan nyttjas för att skapa sådana plattformar. Huvudproblemet innefattar hur arbete ska distribueras och verifieras inom det decentraliserade nätverket. För att avvärja attacker riktade mot nätverket från både interna och externa aktörer används en kombination av incitament och ekonomiska principer. Protokollet använder både en generaliserad blockkedja och andra mer etablerade elektroniska kommunikationssätt. Genom att använda smarta kontrakt möjliggörs tillitslösa överenskommelser mellan deltagare i nätverket.

En referensimplementation av protokollet, vid namn *Zeppelin*, har delvis realiserats i Ethereum-nätverkets generella blockkedja, och har testats på ett mindre nätverk. Referensimplementationen är modulär och demonstrerar möjligheten att använda både en traditionell backend och en blockkedjebaserad backend i decentraliserade applikationer. Genom att använda en blockkedja kan viss applikationsdata och affärslogik lagras och exekveras på en globalt distribuerad virtuell maskin. Referensimplementationen är ännu inte redo för driftsättning, utan ska ses som en teknikdemonstration av det föreslagna protokollet. Att referensimplementationen inte är redo för drift kan tillskrivas det faktum att generella blockkedjor i sitt nuvarande tillstånd ännu inte går att lita på i en produktionsmiljö, eftersom de fortfarande är under utveckling.

Terminology

General terms and abbreviations

ABI
: Application binary interface. A machine-level entry point for a program component.

API
: Application programming interface. A high-level entry point to a software component.

Bitcoin
: Short for the Bitcoin network. For the currency used by the network, see *BTC*.

Blockchain
: A distributed and decentralized data store.

BTC
: (pronounced as Bitcoin) refers to the currency in the Bitcoin network.

Centralized system
: A system with one or more central points.

Cryptographic hash
: A definite one way hash function.

DDoS
: Distributed denial of service attack.

Decentralized system
: A system without any central points.

Ether
: The currency of the Ethereum network. Compare with BTC and Bitcoin.

Ethereum
: A general-purpose blockchain network.

JSON
: JavaScript object notation.

JSON-RPC
: A JSON-based remote procedure call scheme.

MITM-attack
: Man-in-the-middle–attack. An attack where one relays, and possibly modifies, communication between two nodes who believe they are directly communicating to each other.

Nonce
An arbitrary number used only once in a certain (cryptographic) environment.

Peer-to-peer network
A network in which the nodes communicate directly with each other.

Proof-of-work
Regulatory mechanism in a blockchain designed to deter forgery.

REST(ful)
Representational state transfer. An API model.

Smart contract
A piece of deterministic code, run on a blockchain, with which various entities can interact.

Trustless
Literally without trust. Here meant to signify that no trust is neccessary between nodes in the network.

Whisper
An anonymous message passing service on the Ethereum network.

Terms with specific meanings in the thesis

Client
A user paying to perform *work* on the cloud platform.

Verifier
A *worker* that verifies other workers.

Work
An arbitrary piece of software to be run by a *worker*, at the behest of a *client*.

Worker
A cloud platform node performing *work* in exchange of monetary compensation.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Purpose . . . 1
- 1.3 Problem . . . 2
  - 1.3.1 Distribution of work . . . 2
  - 1.3.2 Verification of work . . . 3
  - 1.3.3 Monetary compensation to workers . . . 3
- 1.4 Scope . . . 4

2 Technical background **5**
- 2.1 Bitcoin and the blockchain . . . 5
  - 2.1.1 General idea . . . 5
  - 2.1.2 Transactions . . . 5
  - 2.1.3 Blockchain . . . 6
  - 2.1.4 Proof-of-work . . . 6
  - 2.1.5 Verification . . . 7
  - 2.1.6 Incentive . . . 7
  - 2.1.7 Weakness . . . 7
- 2.2 Ethereum . . . 7
  - 2.2.1 Smart contracts . . . 8
  - 2.2.2 Halting problem . . . 8
  - 2.2.3 Whisper . . . 8
  - 2.2.4 External interfaces . . . 9
  - 2.2.5 Ethereum ecosystem . . . 9
- 2.3 Docker . . . 10
  - 2.3.1 Virtualized containers . . . 10
  - 2.3.2 Security . . . 11
  - 2.3.3 Dockerfile . . . 11
- 2.4 Networking . . . 11
  - 2.4.1 NAT . . . 11
  - 2.4.2 UPnP . . . 12
  - 2.4.3 IGD-PCP . . . 12

3 Method **13**
- 3.1 Implementation workflow . . . 13
- 3.2 Decentralized application platform . . . 13

3.3 Development tools . . . 13
3.3.1 Pivotal Tracker . . . 14
3.3.2 AlethZero . . . 14
3.4 Test environment . . . 14
3.4.1 Hardware . . . 14
3.4.2 Network . . . 15

4 Result **17**
4.1 General protocol . . . 17
4.1.1 Dependencies and assumptions . . . 18
4.1.2 Contract interaction . . . 18
4.1.3 Direct communication between clients and workers . . . 19
4.1.4 Brokering the client-worker agreement . . . 19
4.1.5 Work transfer and execution . . . 19
4.1.6 Work auditing and verification . . . 19
4.1.7 Payment arbitration . . . 20
4.1.8 Security . . . 20
4.2 Reference implementation: Zeppelin . . . 20
4.2.1 Docker . . . 21
4.2.2 System design . . . 21
4.2.3 Smart contracts . . . 21
4.2.4 Deployment . . . 21

5 Discussion **23**
5.1 Method discussion . . . 23
5.1.1 Ethereum client dependencies . . . 23
5.1.2 Implementation environment . . . 24
5.2 Protocol discussion . . . 24
5.3 Reference implementation discussion . . . 24
5.3.1 Whisper . . . 25
5.3.2 Networking . . . 25
5.3.3 Docker and security . . . 26
5.4 Ethical aspects . . . 26
5.5 Environmental impact . . . 26

6 Conclusion **27**

Bibliography **29**

1
Introduction

Cloud computing is an abstraction implying the possibility to store data and execute arbitrary code concurrently and with redundancy. The solutions for cloud computing available today are expensive, centralized, and as shown by recent events [1][2], susceptible to third-party interference, *e.g.* Distributed Denial of Service (DDoS) attacks and espionage by the National Security Agency or other similar agencies. With a decentralized solution, hardware costs can be mitigated and the risk of third-parties eavesdropping or interfering could be lowered or even eliminated.

1.1 Background

Services allowing companies and individuals to purchase cloud computing power and storage from a hosting provider, *e.g.* Amazon and Google [3][4], have existed for some time. However, this places the consumer in a position where trust between the two parties is necessary: trust that the Service Level Agreement is honored; trust that information is kept secret; trust that when the customer deletes information, the information is actually deleted and not secretly kept stored by the storage provider.

Initiatives essentially working in reverse of the above, requesting donations of computation power for scientific purposes have also been around for a while. Examples of these include SETI@home (analysis of radio telescope data) and Folding@home (protein analysis) [5]. These are centralized grid computing projects that have resulted in powerful networks building on voluntary contribution to science.

By combining and decentralizing *a*) the ability to purchase computing power; and *b*) the ability for individuals to participate with their computing power, we propose that a global marketplace of distributed computation power could be created. Such a marketplace would make it possible to buy, sell or even trade computation power, worldwide, with no trust required between parties.

1.2 Purpose

The purpose of this thesis is to investigate how to distribute work on a network of nodes with no central authority, and how to verify that the work has been executed correctly. The key focus is to conduct research in the field of decentralized applications, and to draft a protocol that supports the ideas and features discovered along the way. An effort should be made to create a reference implementation of the protocol and deploy it on a small-scale network. The network must meet the following criteria to be usable:

Transparency. All transactions in the network should be transparent and traceable to show that the system is fair. To achieve this, ledgers must be persistent and auditable.

Resilience. The network must be resistant to attacks from internal and external actors. Such attacks could include disruption of access to the network (DDoS) and attacks that seek to compromise the integrity of ledgers and sabotage the infrastructure of the network.

Trustlessness. Users of the platforms should not be required to trust each other about the details of their agreement when interacting with each other. This should be abstracted away into deterministic code.

1.3 Problem

The overall problem can be summed up as follows: designing a network that lets a client pay to perform work, distributes the work, and pays the workers for their contributions. These are fundamental problems which must be solved in order to fulfill the purpose of the study, and involve a number of challenges in both security and network communication. However, the challenges can be divided into different subproblems that are easier to solve. In this report *client* refers to a user paying to perform work on the cloud platform; *work* refers to some arbitrary piece of software defined by a client and submitted to be run on the cloud platform; and *worker* refers to a cloud platform node which is performing work in exchange of monetary compensation. We address the following three issues:

- How can work be distributed in a decentralized network?
- How can the network verify that workers execute work correctly?
- How can workers receive payment for performed work?

The issues above have so far not been studied in a decentralized context. However, there exists substantial research describing such problems in centralized grid computing infrastructures. To facilitate our research, a theoretical basis has been synthesized using previous studies and reports from the grid computing field.

1.3.1 Distribution of work

A client that wishes to have its work executed must have a way of finding workers. This is traditionally done using a centralized server which maintains a record of all workers and their identities [6][7]. For obvious reasons such a method could not be used in a decentralized network [8]. Therefore, there must be a way to store this information in a non-central location. Furthermore, a uniform way of specifying the work payload is necessary to ensure interface compatibility between clients and workers. There must also be a way to decide which worker(s) that will receive the work. This must be done in a manner that aims for all work to be distributed evenly throughout the network, and requirements from the worker have to be taken into consideration. Whenever a worker receives work, the work must be executed at some

point in the future, implementing weak fairness [9]. Both clients and workers will be interested in metadata, so there must be a way to send and receive information about work and its status to and from clients and workers.

1.3.2 Verification of work

There must exist a uniform way of executing work so that all involved parties agree on what work is to be performed. When a worker has started on a task, there must be some external checking and verification ensuring that the worker is executing the work as expected. The reasons for incorrect execution could be incidental (incompetence, hardware or software failure) or intentional (in an attempt to cheat). This problem is discussed by Chang et al. albeit on a lower abstraction level [10]. Chang et al. argue that a virtual machine must be used to execute work, in order to guarantee the soundness of the completed work across the network.

Redundancy as verification. In distributed computation systems, where tasks are deterministic in nature, some kind of result is expected from participants for each task they have been assigned. The results from the same task performed by different participants can then be compared against each other for proof-checking. For result verification, SETI@home utilizes multiple workers on the same problem, and then compare their results [11]. A solution similar to this cannot be used in our network, since we allow arbitrary code to be executed on workers. The network does not expect any answers from workers, nor is it aware of what kind of application is deployed on a worker node, rendering such an approach useless. In addition, the recent discovery of the $P + \varepsilon$ attack has made this method of verification infeasible in a decentralized context [12]. In this attack, a malicious user offers workers an additional external reward for providing the network an incorrect result.

Mitigating cheating workers. In respect to cheating, *i.e.* workers claiming to have completed work without actually having done so, Yurkewych et al. prove that redundancy in P2P computing is cost effective if and only if cheating workers cannot collude [13]. In a decentralized network, workers could easily create new identities, which makes it hard to know if any two workers collude. It should be noted that Yurkewych et al. assume that it is possible to audit the work of a client. Auditing by external parties could be used if there is a defined way to perform auditing. The notion of credibility-based auditing, which is introduced by Sarmenta [7], would require the overhead of maintaining a list of credible identities, but could indeed be used to solve the issue of workers creating new identities by assigning a credibility reputation to each identity.

1.3.3 Monetary compensation to workers

Monetary compensation would in practice mean a cryptocurrency-based payment due to the inherent trustlessness in such transactions, which is elaborated further in section 2.1. Workers in the network should be paid if and only if they perform work and the validity of said work has been established by network consensus. The payment procedure should be conducted in a way which makes it costly for the worker to cheat, thus making it more attractive to participate honestly than to

cheat. Therefore, combining the payout logic with the validity check on a deeper level could be necessary. If it is more profitable to participate honestly than to cheat, this will ensure that the network is not vulnerable to attacks by internal actors. On the contrary, if the aforementioned profitability can not be guaranteed, the protocol may end up unusable. While the work is being performed on behalf of a client, the payment must be held in some kind of escrow. This is to ensure that the client can not cheat by withholding payment when the worker has performed the work, and to ensure that the worker can not receive payment before the work has been completed.

1.4 Scope

The scope of the thesis is to create a protocol fulfilling the criteria from section 1.2, and attempt to make a reference implementation of said protocol. It is not within the scope of the project to make any applications or programs intended to be deployed on the resulting platform, other than for testing purposes. The protocol should be seen as an open invitation for others to use and develop further.

To ensure that computations are run in an expected manner on a remote agent there must be some kind of virtual environment which allows software to run deterministically and hardware-independent. It does not fall within the scope of the thesis to implement such an environment.

The protocol will face plenty of security risks, and it is necessary to find countermeasures to possible attacks. It is not the main purpose of the thesis to find all possible attacks on the protocol.

2
Technical background

In this chapter we provide a technical background of the concepts which the protocol depends on. In addition, origins and implementations of those concepts will be described and related to the protocol and reference implementation work. Of interest is the cryptocurrency network Bitcoin and its data store called the blockchain. The specific dependencies of the reference implementation, Ethereum and Docker, are also described. A brief summary of computer networking concepts required for the understanding of the protocol and reference implementation is given.

2.1 Bitcoin and the blockchain

Bitcoin sparked a revolution in digital currencies when the white paper was released in 2008. The innovation of the blockchain and its implications is of special interest. All technical details in this section are taken from the Bitcoin white paper [14], unless noted otherwise.

2.1.1 General idea

The purpose of Bitcoin is to provide a financial system where participants can send and receive money without involvement of third parties. Traditionally, these third parties comprise a governmental institution controlling a currency, and one or more commercial banks that transfer money from the sender to the recipient. This is problematic because a third party can perform malicious actions and moreover exert complete control over the process. Bitcoin introduces a decentralized exchange of digital tokens[1], where no middleman is required, nor is any trust between sender and recipient required. The only requirement to use Bitcoin is to have a computer with Internet access. Bitcoin ownership is linked to cryptographic keys, thus, a cryptographic public key is somewhat similar to a bank account. In the following sections, key points of the Bitcoin white paper are discussed. In this report Bitcoin refers to the Bitcoin network, and BTC refers to the currency.

2.1.2 Transactions

Transactions in the Bitcoin network are conducted as follows: If Alice has one BTC and wants to send it to Bob, she will create a transaction stating that she transfers one BTC to Bob. The transaction is signed using her cryptographic private key, and

[1]The intrinsic value of Bitcoin is discussed in-depth by Buterin [15].

then broadcast to the network. An attacker cannot spend Alice's BTC, because the transaction is valid only if it is signed by her private key.

2.1.3 Blockchain

Bitcoin is based around an innovation known as the *blockchain*[2]. Nodes in the Bitcoin network send and receive transactions to other nodes in the network, transactions are then checked for validity and placed together in a block of transactions. To calculate the current state of the network, the whole graph of blocks must be traversed.

A new block is created approximately every 10 minutes, with all transactions that occurred since the last block creation stored in it. Figure 2.1 shows the chain of blocks. Each block contains the hash of its parent; a nonce that proves the validity of the block; and all the transactions in the block. Since any valid transaction is part of some block, and all valid blocks are chained together, the blockchain can be seen as a distributed ledger, and proof of the true state of the network. Once the block has been created, any subsequent changes to a block will invalidate the nonce, and thus invalidate the nonces of all child blocks.

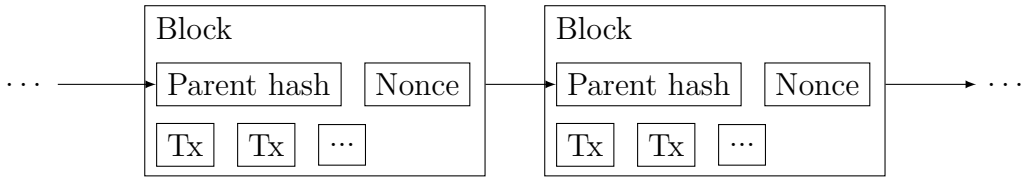


Figure 2.1: Visualization of blocks in the Bitcoin blockchain.

2.1.4 Proof-of-work

Like the name proof-of-work suggests, it is meant to be a regulatory mechanism making it difficult to falsify blocks. As stated above, the nonce is used to validate the integrity of a block, but also acts as a way to show that a sufficient amount of computing power has been consumed. The proof-of-work process is implemented by finding a certain hash value related to the block content, which is impossible to precompute. Thus the nonce of the block must be iteratively chosen at random until a low enough hash has been calculated. It is thus simple to verify that the nonce is correct, but hard to generate a valid one. Specifically, the nonce is created by calculating the hash of all the contents in the block with a function returning a hash value lower than a network-wide set value, which is determined by the current difficulty level of the network. The difficulty level is a mechanic used to control how long, on average, the nodes of the network must work before a block is found. Once every week the difficulty is adjusted, to match the computing power available in the network. This is tuned to ensure an average block rate of approximately one block every 10 minutes.

[2]The grammatically correct "block chain" has not gained widespread popularity.

2.1.5 Verification

The blockchain solves a fundamental problem of transferring value online, namely that data (for instance money) could be copied and sent to multiple recipients. Alice could spend the same BTC twice by signing two different transactions. However, only one of these transactions would be accepted into a block, after which the other transaction would be considered invalid, and thus would not be accepted into any block. In the same way, an overspending transaction would also be considered invalid.

2.1.6 Incentive

In order for the network to fully function, nodes in the network must provide the verifying calculations, *i.e.* the proof-of-work, to continuously assert the integrity of the blockchain. Without incentive, it is unlikely that peers would share their computing power to benefit the network. In the Bitcoin network, whenever the proof-of-work for a block is found, the worker is rewarded a number of BTC. The current reward is 25 BTC, however this reward is halved every four years. Offering a reward for the verification process thus leads to a more secure and robust network.

2.1.7 Weakness

The most widely known weakness in the blockchain structure is often referred to as the 51%-attack. By default, nodes accept the longest blockchain as the representation of the network state, and the longest blockchain is always the one that has the most raw computing power behind it. Thus, if an attacker has control of more than 50% of the computing power in the network, the attacker can take control of the blockchain. This makes it possible for the attacker to void transactions and possibly modify recent transaction history. If an attacker manages to modify a block in an illegal way, and computes enough subsequent proof-of-work to make this blockchain the longest, other nodes will be forced into accepting that chain, as they have no way of outpacing the attacker. This situation is in practice very unlikely, since massive computing power would be required in order to overthrow the Bitcoin blockchain.

2.2 Ethereum

Following the advent of Bitcoin and the blockchain, a number of initiatives were started to explore and utilize the power of the blockchain within a distributed and decentralized application context. One example is Namecoin, which uses the blockchain to store domain names. As a result of each project aiming to solve one specific problem, segmentation in the cryptocurrency projects ensued, with much duplication of effort. The Ethereum project was started to unify the open source blockchain efforts and provide a general purpose blockchain protocol which allows running arbitrary code in the blockchain. This means that any type of features can be programmed onto this protocol. Code is packaged into so called smart contracts, executed on the Ethereum Virtual Machine (EVM) [16]. The Ethereum net-

work is built on top of DEVP2P, a peer-to-peer network between all nodes running Ethereum-compatible software.

In the Ethereum network, the cryptocurrency rewarded for mining blocks is called Ether. Ether ownership is very similar to the ownership of BTC, where Ether is linked to a public key. However, there is one additional aspect that is very important, namely that smart contracts can also own Ether. In contrast to the Bitcoin network, this allows code run in a trustless environment to have complete control over its funds. Transactions in the Ethereum network can, besides transferring Ether, also make method calls to smart contracts.

2.2.1 Smart contracts

Smart contracts are a way to move and control assets using code to set up and enforce arbitrary rules. Contracts can be defined in the three programming languages listed below. All of these programming languages are Turing-complete. The programming languages can be used to define smart contracts and must be compiled to EVM byte code before they are incorporated into the blockchain.

- Solidity (C- and JavaScript-like)
- Serpent (Python-like)
- LLL (Lisp-like)

Smart contracts are inserted into the blockchain by creating a transaction containing the EVM code. The resulting contract will receive a unique hexadecimal identifier that can be used for subsequent contract interactions. Such interactions would be the typically object-oriented task of getting and setting values, and invoking methods. In addition, a contract can create other contracts on the Ethereum blockchain, much like how an object can instantiate other objects in object-oriented programming.

2.2.2 Halting problem

The notion of a Turing-complete language executing in a blockchain raises the halting problem. In a Turing-complete language, there is no way to know whether a particular program will halt or not [17]. To ensure that each contract interaction does halt, every Ethereum virtual machine instruction has a price, which must be paid by the requesting party [18]. Therefore it is possible to argue that the EVM is *quasi*-Turing–complete in practice.

2.2.3 Whisper

The messaging protocol Whisper is used to send anonymous messages over the Ethereum network. A naive solution for a messaging protocol for the network would be to store messages in the blockchain, however, this is infeasible because it would be too expensive to send such messages. Whisper solves the problem for short- to medium lived messages. This is done by enabling users to send messages to topics

or with a public key as recipient. The origin node sends a Whisper message to each of its connected peers. Whenever a Whisper message is received, the process is repeated until the time to live of the message has expired. Before the initial broadcast of the Whisper message, a small proof-of-work must be completed, in order to prevent flooding of the network.

2.2.4 External interfaces

An Ethereum client (either local or remote) can be accessed by two APIs. The lower level API, the Ethereum generic JSON-RPC, is a remote procedure call scheme suited for fine grained control of the host client [19]. On the higher level, there exists a JavaScript API built on top of the JSON-RPC API [20]. The JavaScript API is targeted at usage in web application frontends with a backend either completely or partially stored in the Ethereum blockchain, with a reference implementation called web3.js being actively developed.

For communication with the blockchain, the expected format is binary. The Ethereum contract ABI (Application Binary Interface) specifies the binary format [21]. For the end user, or developer, the ABI is abstracted away by higher level constructs, such as the JavaScript API, which handles the conversion from high level objects to binary data.

2.2.5 Ethereum ecosystem

To use the Ethereum network, a client connected to the DEVP2P network is required. There are a number of different clients, and client environments, listed in Table 2.1. Note that both command line and graphical clients are provided. The primary Ethereum client implementations are in Go and C++, with a secondary implementation in Python and a peripheral Java implementation [22][23][24][25]. These clients serve as the interface to the Ethereum decentralized application platform and the Whisper anonymous message passing service.

Table 2.1: Ethereum client list by language.

Language	GUI client	CLI client
Go	Mist	geth
C++	AlethZero	eth
Java	studio	core
Python	N/A	pyeth

Under the Ethereum project umbrella, a whole ecosystem for decentralized applications is being developed. To deliver decentralized applications to end users, the preferred method is to build frontends in standard web technologies, such as JavaScript and HTML, which most end users and developers are familiar with. Since the identity of the user is linked to their Ethereum public key, users never need to sign up or create accounts in Ethereum-based decentralized applications, since their public key uniquely identifies them. The frontend interfaces with an Ethereum client to interact with the underlying network peers and the business logic stored in the

smart contracts associated with the application. In Figure 2.2, the full application stack is listed. From a development standpoint, this invites to rich client applications and thin servers, where in fact most web content can be served statically, since the application data and business logic is stored in the blockchain. Note the contrast to traditional web application architectures, where data and business logic is available only to and through the application server. With a decentralized application, the data and business logic is available and auditable by all users of the application.

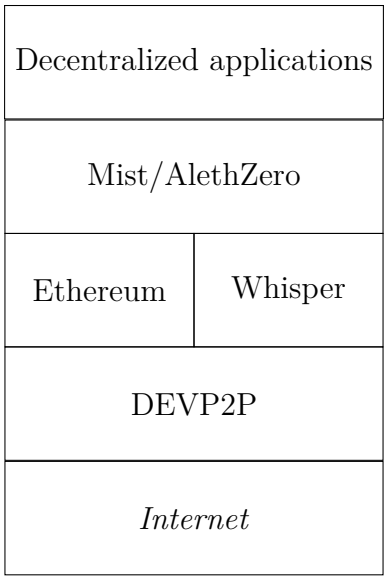


Figure 2.2: Ethereum application stack.

2.3 Docker

Software stacks today are generally quite complex. A website, for example, is most often much more than a simple HTML document. To generate the HTML document, there could be numerous programs and services running on the server side, *e.g.* the web server, components encapsulating business logic, rule engines, databases, document storage and so on. To deploy all these services manually, on different platforms, would be time consuming and error prone. With Docker, such processes can be containerized, which allows the application to be run in a virtual-like manner on the Linux kernel, without the performance overhead of running a virtual machine [26], simplifying the deployment and management of large software infrastructure.

2.3.1 Virtualized containers

Docker uses virtualization techniques to create isolated environments called containers which can be executed as if they were a runnable file. There are two distinct advantages for this. The first one is that since the containers are a full application packaged with its dependencies, it comes precompiled and is ready to run on any architecture that supports Docker (and is on the same architecture the application was compiled for). The other advantage is that a containerized application

is isolated from both the system it is running on, as well as other containerized applications running on the system. If the code were to run directly on the host system, it would have been exposed to a number of exploits [5]. Containerizing and virtualizing mitigates the immediate threat of such exploits.

2.3.2 Security

An important factor when volunteering to run arbitrary code on a private machine is security. The software could potentially corrupt the data on disks, or steal all the data. Docker containers are executed in a separate environment from other containers and the host machine, meaning that it can not know about or interact with the outside world [27]. To communicate with the outside world, certain ports can be exposed on the container, and will be mapped to the host machine. Thus, network communication can only occur in an expected and controlled way.

A containerized application can be run with restrictions on CPU and memory usage, to prevent memory leaks from propagating to the host system.

2.3.3 Dockerfile

Docker images are defined by a format called Dockerfile [28]. The easiest, and the most common method of building a Docker image is to use one of the readily available base images, provided by software distributors and developers such as Nginx, Ubuntu and PostgreSQL to name a few. A Dockerfile can be created by composing it on top of an existing pre-configured base image for simplicity. The Dockerfile serves as the blueprint for the Docker image, being the realization of the Dockerfile.

All Docker images are saved in the local Docker repository, from which images can be exported and distributed for deployment on other machines. A running instance of a Docker image is called a Docker container. Several Docker containers of the same image may be run concurrently without interfering, since they are containerized.

2.4 Networking

Communicating between devices over a local network is relatively simple, but for connections via the Internet, a device behind a gateway can not be directly reached by a remote device. There must be a path through the gateway mapped to it, via a method called NAT [29]. The service then becomes available externally via the external IP address and a port number.

2.4.1 NAT

Network Address Translation is a method commonly used to map any device on an internal network to an external network like the Internet [29]. It works by storing an internal port number paired together with the local IP address of the local device and the external IP paired together with an external port on the gateway device of

the network. When packets are received by the gateway with the combination of an external IP and a NAT port, they are modified and forwarded to the internal device.

2.4.2 UPnP

Universal Plug and Play, abbreviated UPnP, is a network protocol that allows connected devices to discover the presence of other such devices and establish a connection to each other on a network in order to share data, communicate and allow media services to interoperate [30]. UPnP port forwarding can be implemented in several ways, such as IGD-PCP (which will be described) and NAT-PMP.

2.4.3 IGD-PCP

Internet Gateway Device - Port Control Protocol, IGD-PCP, is a NAT port mapping protocol which is supported by many routers. Many applications, such as Ethereum, require the ability to send and receive data from the Internet. In cases where there is a NAT enabled router between the sender and recipient, NAT traversal must take place in order for communication to work [31]. As seen in the list below, IGD-PCP enables a client computer behind a NAT-enabled router to find out the external IP and then add and/or remove port mappings to the computer.

- Learn the public (external) IP address
- Requesting for a new public IP address
- Enumerate existing port mappings
- Add and remove port mappings
- Assign lease times to mappings

3
Method

We shall here describe our methodology for the development process of the protocol and the reference implementation, and provide backgrounds and motivations for the third party software used. The test network used for the reference implementation is also described.

3.1 Implementation workflow

All produced software artifacts and their source code have been open source throughout the development process, so that others might benefit from the research, especially since this technology is in its cradle. Dependencies on third party software should only be made following an assessment of the stability, security and longevity of the software. Based on these principles, the stability of the foundation upon which the produced artifacts depend can be maximized. Exempt from this principle was the Ethereum software, which was under development throughout the entire length of the study.

3.2 Decentralized application platform

An existing blockchain is used due to the inherent security of a larger network as described in section 2.1.7. Implementing a separate blockchain, designed specifically for our purpose, would be out of scope for the thesis and would not bring any advantages. Bitcoin has the most widely used blockchain, but it is not designed to support building applications upon. Thus, we have opted to utilize the Ethereum general-purpose blockchain, which focuses on providing a full stack for decentralized applications. Ethereum is currently the only project aiming to create such a platform.

3.3 Development tools

The following tools have been used to provide a more efficient development process and a refined project planning structure.

3.3.1 Pivotal Tracker

In order to aid the agile development process Pivotal Tracker is used. Pivotal Tracker is a story-based planning tool which allows members of the team to add stories to a virtual project board. It is based on agile development methods, which suits the team. Pivotal Tracker provides several story logs: tasks for the current sprint, a backlog, and an icebox for stories that need grooming [32]. Stories are automatically arranged in the backlog according to their estimated size and the projected sprint velocity. Each story item can be further elaborated by a number of subtasks related to it.

3.3.2 AlethZero

AlethZero is a proof of concept graphical client for Ethereum [33], implemented in C++ and Qt. It is used as a development environment for Ethereum-based decentralized applications and provides rich access to the blockchain and the DEVP2P network. For debugging, or manual blockchain interaction, it is possible to make Ether transactions, contract creation transactions or contract interaction transactions from the graphical user interface. A graphical user interface is also available for Whisper. AlethZero exposes the JSON-RPC API, making it possible to use it as a backend for decentralized applications.

3.4 Test environment

A test environment was chosen and set up in order to test the implementation of the cloud computing platform, with multiple actors. In this chapter detailed information about the system is provided.

3.4.1 Hardware

To test the project, a computer with sufficient processing capabilities to host a simple and small blockchain, as well as a web server when acting as a worker, was required. The Raspberry Pi 2 Model B was chosen because it is a cheap and viable computer. The hardware specifications are listed in Table 3.1.

Table 3.1: Raspberry Pi 2 Model B Specifications [34].

Processor	900MHz quad-core ARM Cortex-A7 CPU
Memory	1,024 MB LPDDR2
I/O Ports	4x USB, RJ45, microSD, 40-pin GPIO
A/V Ports	HDMI, RCA video, 3.5 mm audio
Networking	10/100Mbit Ethernet
Storage	microSD card slot

3.4.2 Network

During development, for testing and researching purposes, a local network consisting of three Raspberry Pi 2 computers was deployed. This network has served as the backbone for distributed compilation, a private blockchain network, local Ethereum test network nodes, and client as well as worker emulation for the reference implementation called Zeppelin.

Compiler network. Since Ethereum is still unfinished software, it must be updated regularly. This takes a large amount of processing power every time it is done, and while the Raspberry Pi 2 is a capable machine for testing purposes, the several hour long task of recompiling Ethereum every other day is too much of an obstruction to the work flow. In order to speed up the process a distributed compiler network was deployed at an early stage. To facilitate this the hardware is networked together via an Ethernet switch, enabling fast communication between all the nodes as well as the Internet when required.

Private blockchain. While testing and developing the reference implementation, some level of control over the blockchain was required. Primarily because both running as well as testing any software in the Ethereum network requires the use of the Ether cryptocurrency. For this reason, a local Ethereum blockchain was deployed during the early stages of development.

Nodes in the public blockchain. The test environment was eventually moved to the public blockchain, to have a complete mirror of the production environment while developing.

Client and worker emulation. Since access to decentralized applications on the Ethereum network can be provided through a web application, the clients running on the Raspberry Pi 2 computers can be accessed remotely and used as test slaves. The most simple way to achieve this, is to expose the JSON-RPC port, and setting the web application to use the remote JSON-RPC as its data source.

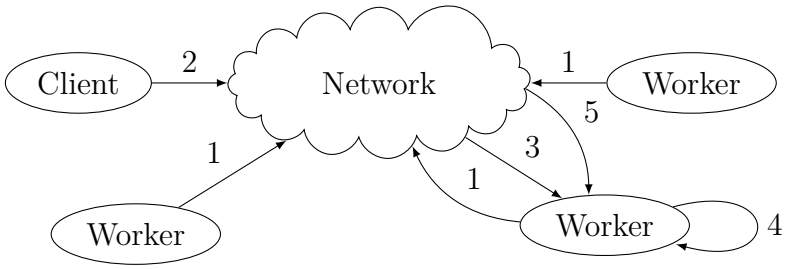
4
Result

Our proposed solution is a general protocol, which fulfills the demands of the proposed network outlined in section 1.2 and solves the underlying problem described in section 1.3. In addition to the protocol, a reference implementation has been developed and tested on a small-scale test network.

4.1 General protocol

The network connects clients, who pay to perform work, with workers, which are paid to perform work. In order for the network, and its cloud computing platform, to thrive, it relies on an free market where the supply of workers meets the demand of clients. Any person with a computer could register to become a worker and receive payment for the work they perform, and conversely, any person with the desire to execute arbitrary code can do so, provided they can pay for it. The high level flow of workers and clients within the network is shown in Figure 4.1.

We propose that a blockchain with capability to store and execute code can be used to satisfy the requirements specified in section 1.2.



1. Workers register to the network
2. Client sends work and payment to the network
3. Worker receives work
4. Worker performs work
5. Network confirms work and sends payment to worker

Figure 4.1: Proposed simplified network state diagram.

4.1.1 Dependencies and assumptions

We assume that there exists some external blockchain in which the network will store its data and that it provides trustless arbitration between the involved parties. The term contract is used as a shorthand for smart contract, here defined to be at least capable of the set of *quasi*-Turing–complete actions described in section 2.2.2. We also assume that two peers can communicate anonymously, given that each peer has access to the public key of the other peer. This will be referred to as private message passing. Finally we also assume there exists a method to package work so that it is easy to distribute from clients to workers, and for workers to perform. Note that the protocol is platform agnostic, in spite of the assumptions made and the dependencies required.

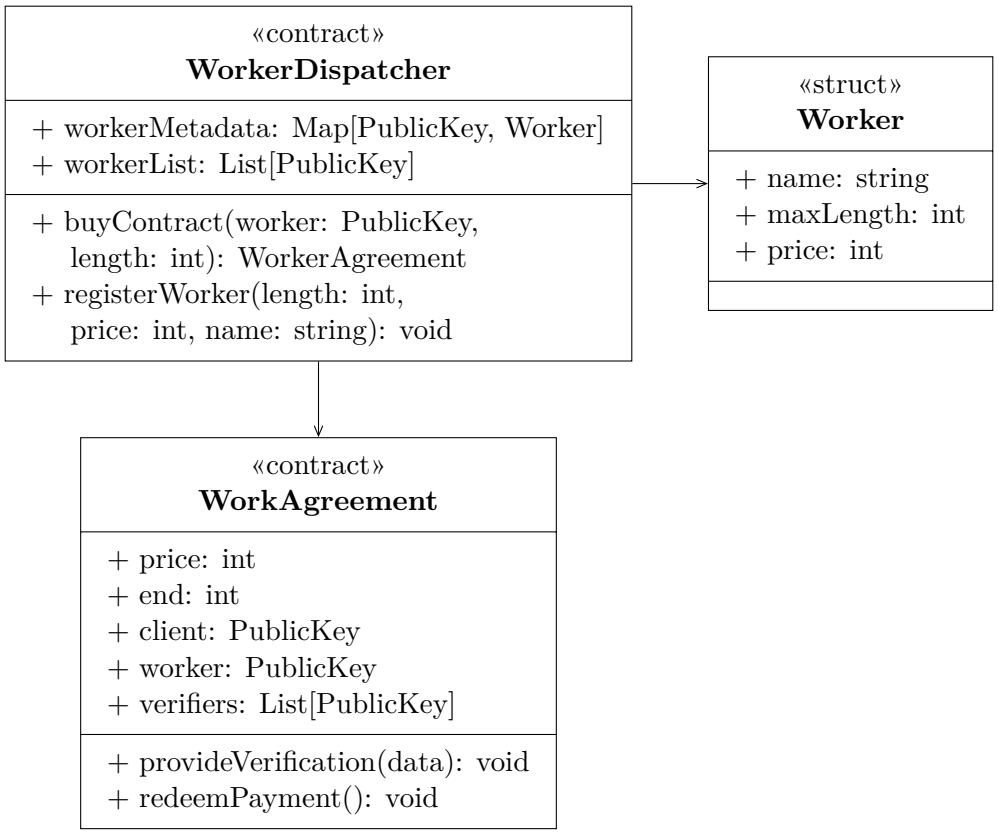


Figure 4.2: UML diagram showing the proposed smart contracts. The type PublicKey refers to either a cryptographic public key, or the cryptographic hash of such a key.

4.1.2 Contract interaction

Communication within the network consists of actions which change the state of at least one of the associated contracts in the blockchain. Contracts are used for *a*) storing data that must be persistent; *b*) arbitration between worker and client; and *c*) to hold monetary value in escrow. For these tasks, two contracts are used;

WorkAgreement and WorkerDispatcher, the later being the main contract. The contracts are shown in Figure 4.2.

WorkAgreement. To provide interaction between the client and the worker for a work instance, the contract WorkAgreement is used. The contract holds the value paid by the client for the work to be done in escrow until the work is finished and holds information about auditing reports from the external verifiers of the work. Once the worker has performed the work as described in the contract, the value held in escrow can be released to the worker.

WorkerDispatcher. As the main contract of the protocol, this serves as the entry point. WorkerDispatcher provides infrastructure for workers to register their interest to perform work, by calling *registerWorker*. Clients use this contract to buy WorkAgreement contracts. This is done by calling the method *buyContract* with desired parameters. The WorkerDispatcher creates a WorkAgreement, assigns a worker, and a given number of verifiers.

4.1.3 Direct communication between clients and workers

For the communication between specific clients and workers, private message passing is used. When transferring data from the client to the worker, encrypted TCP/IP traffic is used. The hosted data provided by the worker is then accessible via the appropriate application protocol.

4.1.4 Brokering the client-worker agreement

As soon as a client has received a WorkAgreement from the WorkerDispatcher, the client initiates communication with the assigned worker through private message passing. The process starts with the client sending the WorkAgreement contract address to the worker, after which the worker proceeds to check if it agrees with the contract. When the verification is done, the worker sends a message via private message passing to the client with the IP and port to be used to transfer the packaged work (explained in section 4.1.5). Upon receiving the work, the worker starts to deploy the packaged work. When the work is ready to be accessed, the worker sends a message back to the client specifying the IP and port for external access.

4.1.5 Work transfer and execution

When the contract negotiation is finished, the work, defined as code executable on the worker, must be transferred to the worker. The worker is placed into a receiving state, and the client can proceed to send the work. Once the worker has received the work, it proceeds to execute it, stopping once the contract has been fulfilled.

4.1.6 Work auditing and verification

In order to know that a worker is performing a given work, other workers are assigned as verifiers. These verifiers are given a time frame in which they are required to report checked parameters of the given worker to the WorkAgreement contract.

This is done through the *provideVerification* method, which is callable only by verifiers. The most basic check that could be done is simply for the verifier to assert that the worker is providing the given work through the IP and port that it has communicated. This does however only check uptime of the given worker, and not that the work has been correctly performed. More rigid verification could be possible if the verifiers were able to make a more specific request to the worker, but since the verifiers do not know anything about the work to be done, the client must provide such specific request information.

4.1.7 Payment arbitration

When a WorkAgreement is created it must also be provided with an appropriate amount of payment for the given work. The value of this payment is then held in escrow by the contract until some specified conditions in the contract have been met. These conditions consist of verifications that the verifiers have performed and checks that the worker has performed for the given amount of time. When all conditions are met the worker can call the *redeemPayment* method in the WorkAgreement and the contract will release the payment held in escrow to the worker. In a similar fashion, all the verifiers of a WorkAgreement can withdraw payment for their completed services, although their reward is significantly smaller. Any calls to *redeemPayment* by a party not entitled to receive payments will be ignored.

4.1.8 Security

The protocol itself can be seen as secure in most aspects as it mostly runs on top of a blockchain. Security concerns can instead be found in that individual workers can be attacked. Workers provide their services through regular HTTP connections and is therefore vulnerable to *e.g.* DDoS-attacks and man in the middle attacks (MITM), just as regular service providers are. The protocol does not take this into account and thus only reward workers that have successfully provided results. However if the protocol is implemented correctly only the client and verifiers can know the IP address of the worker if the client chooses not to share it.

4.2 Reference implementation: Zeppelin

The reference implementation, called Zeppelin[1], is a web application for end users, which is dependent on the Ethereum network. For computational integrity, the client constructs a Docker image which is sent to the worker. The implementation is built around the concept of a thin server, and a fat client. It is worth noting that there are two backends with which the frontend communicates; the web server, acting as the *traditional* backend; and the smart contract interaction through Ethereum, acting as the secondary backend for all blockchain-based interactions. Zeppelin is not a complete implementation of the protocol described in section 4.1. Rather it solves only the problem of distributing work in the network. However it is possible to build

[1]Because a Zeppelin navigates above the clouds with grace.

upon Zeppelin to implement the remaining features of the protocol. All source code for the reference implementation is available from https://github.com/dccp.

4.2.1 Docker

To package work to be sent between clients and workers in the reference implementation, Docker was chosen. Since Docker provides a way for applications to be containerized with their dependencies and distributing image files is simple, it is a good fit.

4.2.2 System design

The system design is characterized by high modularity, allowing for reuse or reimplementation of most modules within the solution stack. In Figure 4.3, the component diagram for the full stack is shown. The only strict dependency is on Docker, all other components can be substituted.

Zeppelin frontend. For the simplicity of the end user, all interactions are made in a web interface, in the form of a JavaScript client application. The frontend interacts with two backends to display data: to communicate with the Ethereum client over JSON-RPC, the library web3.js is used; to manage Docker the REST API in the Zeppelin backend is used.

Zeppelin backend. This module acts as the *traditional* backend, serving the static client web application and acting as a RESTful server for user interactions with Docker, through the Docker transfer module.

Docker transfer. Works as both client and server for sending and receiving gzipped Docker images and running the containers. When the worker is placed in the receiving state, a Docker transfer server is spawned on the worker, waiting for a connection from the client. Once a connection is made the client compresses the desired Docker image and sends it to the worker, which imports the image into its own Docker repository.

4.2.3 Smart contracts

The smart contracts used for business rules in the blockchain are realized in the C- and JavaScript-like Ethereum-specific language Solidity. These contracts adhere to the specification in section 4.1.2 to the extent required to implement the distribution of work task. Since a private message passing is mandated by section 4.1.1, it has been implemented in the WorkAgreement contract, which means that the contract will be responsible for relaying the necessary connection information between workers and clients. With this solution, privacy can not be ensured, since the information is freely available in the contract storage.

4.2.4 Deployment

The Zeppelin application stack (the frontend, the backend, and Docker transfer) is bundled together in a Node.js application. When deployed, it can accommodate a client, a worker, or both without any special configuration. As shown in Figure 4.3,

the reference implementation is dependent on a Docker instance to be running, as well as access to an Ethereum client over JSON-RPC.

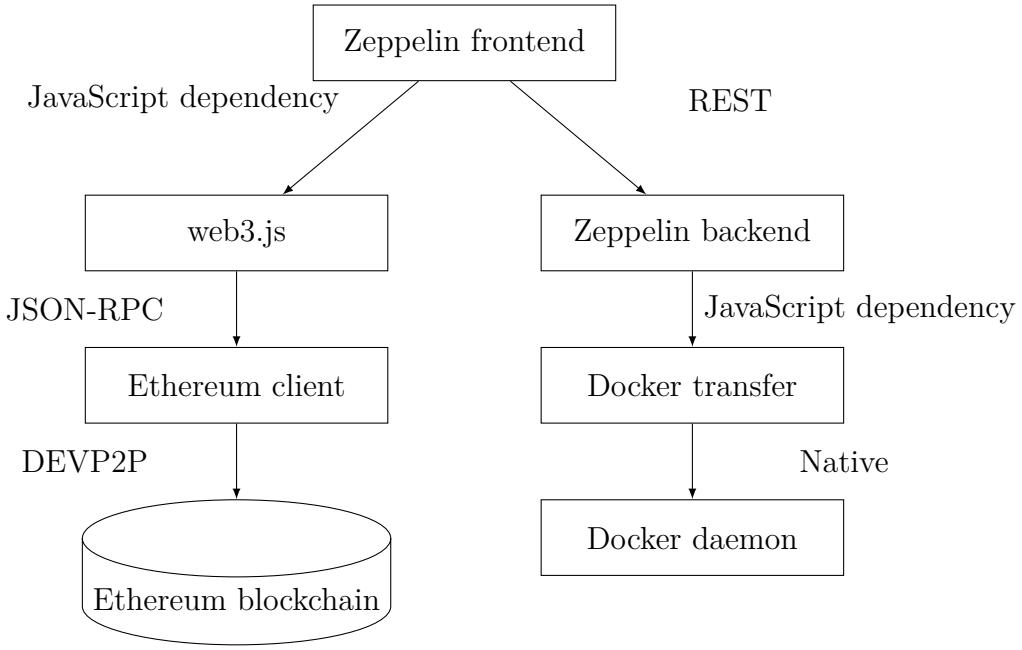


Figure 4.3: Component diagram of the full Zeppelin stack.

5
Discussion

The result of the study is a protocol for a decentralized cloud computing platform where a free market is used to distribute work throughout the network. The protocol itself has no single point of failure, or any external authority which might revoke access to it. However, the workers in the network can still be individually attacked, but it can be argued that workers have the motivation to mitigate such attacks since their payment is dependent on work performance. The network achieves decentralization because *a*) the specification and reference implementation are both open source; *b*) each end user retains a copy of the blockchain in which the smart contracts and data of the application are stored; and *c*) each end user runs the application locally.

In the introduction, three basic requirements for the network were listed. The requirements call for the network to be transparent, resilient and trustless. By virtue of the open and distributed data store of the blockchain, the network can be deemed to be transparent. Resilience is achieved on the data store level by the Ethereum network, and on the work execution level by incentives offered to workers. The inherent trustlessness in the blockchain and smart contracts allows parties to agree upon work and payment. Thus, we argue that these requirements have been met.

5.1 Method discussion

To alleviate further research within the field, and to allow future studies to build on the conducted thesis work, the methods will be briefly discussed. A central point is that the thesis work suffered from unnecessary overhead due to dependencies on unfinished software.

5.1.1 Ethereum client dependencies

Ethereum is the only platform currently providing a functional implementation of a general purpose blockchain. Therefore, it was chosen for the reference implementation. At the inception of the study, three different Ethereum client implementations were widely used: the C++ based client cpp-ethereum; the Go based client go-ethereum; and the Python based client pyethereum. All of the clients were at the time developed in parallel, stepping towards milestones implementing different aspects of the Ethereum specifications. The different clients were not always completely interchangeable, which was cumbersome while testing new client features on the different platforms. Since the stable releases were rapidly becoming out-

dated, the implementation was dependent on bleeding edge builds of the C++ and Go clients. As a result of this, large amounts of time was spent waiting for regression errors to be addressed in certain clients, and for cross-compatibility or workarounds between the two clients to support the development efforts. Thus, the dilemma was to either stay at a stable client version without necessary features — or to stay on the potentially broken bleeding edge. This problem can be completely attributed to the fact that depending on unfinished pre-released software has detrimental effects on the expected productivity.

5.1.2 Implementation environment

The initial solution was to implement the reference implementation in a command line Python client, using HTTP requests to access the JSON-RPC API provided by Ethereum. This solution was abandoned because the JSON-RPC API expects input in the form of the Ethereum ABI (Application Binary Interface), which means that the client would have to supply the binary format. Implementing the ABI would be out of scope and too time consuming for the study. Thus, the web3.js library, which implements the Ethereum JavaScript API, acting as a higher level bridge to the Ethereum client. Changing from Python to JavaScript resulted in also changing from a command line client to creating a web application.

5.2 Protocol discussion

While the protocol addresses all three problems presented in section 1.3, it does so on different abstraction levels for the different subproblems. The most thoroughly described and concrete part of the protocol is the part regarding distribution of work. This matter is described in a sufficiently detailed way to make it simple to implement.

The verification of performed work is described on a more abstract level, omitting various implementation details that could prove to be necessary in order to make the verification process run reliably and to minimize trust between nodes. The approach described should however provide a solid ground from which the verification process can be further elaborated.

Regarding monetary compensation, the problem description is quite short. This fact is reflected in the protocol. However, this does not mean that there is any significant information missing. In any sufficiently advanced smart contract system it should be trivial to retain value in a contract and to release said value to specific users if and only if certain conditions are met.

5.3 Reference implementation discussion

The reference implementation, Zeppelin, does not fulfill the whole protocol specification. One of the main factors behind this is that the Ethereum platform is still under heavy development. Throughout the study, it has been unreliably to test new features, often with many regression errors and breaking API changes. This makes

it complicated to make any steady progress on many protocol features. Instead, focus was shifted to assure that the distribution of work issue was solved as well as possible.

Since the decentralized technology is emerging, demonstrations and proof of concepts are key tools to achieve user and developer participation. In that sense, Zeppelin is not an optimal proof of concept for the technology due to the fact that it still requires a regular backend. However, it should still be viewed as an accomplishment, because we show that both backends can, in fact, interoperate. This can be seen as the greatest virtue of Zeppelin — it is truly a proof of concept.

5.3.1 Whisper

The private message passing described in section 4.1.1 was meant to be fulfilled by Whisper, the peer-to-peer anonymous messaging protocol running on the Ethereum network. Whisper was to be used in order to exchange information between a client and a worker, such as IP addresses and ports. Since Whisper only requires a public key and/or a topic in order to initiate the communication channel, it can be used as private message passing. However, the Whisper project has not yet been developed fully as of 18 May 2015. Initial tests using Whisper as private message passing gave insufficient performance to be able to function as expected. As a result of the decision to work around Whisper, and instead store more information in the smart contracts. Due to this sacrifice, some of the privacy and security measures mandated by private message passing were sacrificed in order to achieve a working reference implementation.

Security is affected in the following way: in the reference implementation, all IP addresses and ports are accessible via smart contracts for anyone with access to the blockchain. If an attacker has all IP addresses for every node (both worker and client), it is trivial to DDoS the nodes in the network. This is particularly dangerous in since, at present, the network is very small.

5.3.2 Networking

The Internet is a large complicated structure, a network of networks. Negotiating communication between hosts is seldom trivial, due to the way the Docker transfer module works, workers behind routers need to have one or several ports forwarded to be able to accept direct communication from a client. As all the test units used throughout the study were hosted on a Local Area Network (LAN), under a single private IP address, this caused problems with Network Address Translation. In a centralized system this could be solved by using the centralized server as an intermediary, where both the client and worker initiates communication with the centralized server, creating a network tunnel allowing communication between the two. But since Zeppelin is designed to be decentralized this is not possible. The only viable solution is to have the client connect directly to the worker, and this requires an open port.

The initial solution to this problem was to use either UPnP or NAT-PMP for internal port mapping. This would, in theory, solve the problem. Unfortunately,

due to time constraints and problems during the implementation this feature did not reach a working state. The only way to test UPnP and/or NAT-PMP is via a LAN behind a NAT-enabled router. Since all development and testing was conducted remotely from a network with public IP addresses and no NAT-enabled router, there was no compelling way to ensure this functionality. Instead of continuing on the UPnP solution, it was decided that manual port forwarding was sufficient for the reference implementation.

In the future, this solution could be extended to become completely network environment agnostic.

5.3.3 Docker and security

The security that Docker offers is only on the host level. Thus, if a client is trying to compromise the integrity of a worker by instructing it to run malicious work, Docker will deter such attacks. Since Docker is an external dependency the end user is required to keep it up to date independently from Zeppelin, meaning that no responsibility can be assumed for security flaws in external software. Should the IP address of a worker become known by external parties, it could by used to initiate DDoS attacks or other similar attacks against the specific worker. This is not something that can be handled by Zeppelin — instead, an effort is made to work proactively by shielding all such identities behind public keys and private message passing.

5.4 Ethical aspects

Since the network is fully anonymous and decentralized, there is no way to control the legal or ethical aspects of the work offered by clients. Thus, clients could requisition computing power to perform morally dubious or illegal actions. While the authors do not condone such actions, any sort of strike against it would compromise the openness of the platform and defeat its purpose. Since we merely have created a tool, we argue that it does not fall upon us to decide how end-users utilize that tool.

5.5 Environmental impact

Not much research has been conducted on the environmental impact of decentralized networks, perhaps because it is very hard to foresee how nodes in the network will behave and how they will consume power. However, the Bitcoin network has proven to outperform traditional monetary transaction networks by far when comparing carbon dioxide emissions [35]. It is not clear whether this is applicable when comparing arbitrary decentralized networks and their centralized counterparts, but since McCook's work is describing decentralization as more efficient, it can definitely be argued that decentralized applications will have an overall positive impact on the environment when compared to centralized dittos.

6
Conclusion

In the introduction, three research questions were stated: How can work be distributed in a decentralized network? How can the network verify that workers execute work correctly? How can workers receive payment for the work they perform? To answer these questions, solve their associated problems and create a robust and reliable network for arbitrary cloud computation, both a general protocol and a modular reference implementation have been created. While the reference implementation only handles the problem of distributing work in the network, the proposed protocol should be seen as a generalized attempt at solving all three problems.

In its current state, the reference implementation is unsuitable for use in a production environment due to security concerns. Even so, it is completely capable as a proof-of-concept for a decentralized application running in a general-purpose blockchain. In the future, reference implementation could be extended and further refined to implement the full protocol, which however is dependent on the advancement of the Ethereum decentralized application stack. A more rigid specification and verification of the protocol could be necessary to achieve a fully functional system.

The aim of the thesis is to eliminate the third party present in cloud computing services, thus reducing the need for trust. It is possible that the ongoing trend of such decentralization could be the next major advancement in the software sphere, and we are eager to both be a part of and watch it happen.

Throughout the study it has become obvious that the underlying general-purpose technologies currently available for decentralized applications are still in a very early development stage. Before a platform for decentralized cloud computing could be fully realized in such an environment, those dependencies must become more stable.

Bibliography

[1] "Computer says no; Cyber-attacks," *The Economist*, vol. 407, pp. 63–64, 2013.

[2] G. Greenwald, "NSA Prism program taps in to user data of Apple, Google and others," *The Guardian*, 2013, accessed 27-May-2015. [Online]. Available: http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data

[3] Amazon Web Services, Inc., "Amazon EC2 Product Details," 2015, accessed 27-May-2015. [Online]. Available: http://aws.amazon.com/ec2/details/

[4] Google, Inc., "Overview of App Engine Features," 2015, accessed 27-May-2015. [Online]. Available: https://cloud.google.com/appengine/features/

[5] E. J. Korpela, *SETI@home, BOINC, and Volunteer Distributed Computing*, ser. Annual Review of Earth and Planetary Sciences. Palo Alto: Annual Reviews, 2012, vol. 40, pp. 69–87.

[6] D. P. Anderson, E. Korpela, and R. Walton, "High-performance task distribution for volunteer computing," *First International Conference on e-Science and Grid Computing, Proceedings*, pp. 196–203, 2005.

[7] L. F. G. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, vol. 18, no. 4, pp. 561–572, 2002.

[8] P. Baran, "On distributed communications networks," *IEEE Trans. Prof. Commun.*, vol. CS-12, p. 1, 1964.

[9] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2nd ed. Addison-Wesley, 2006, pp. 23–24.

[10] B. Y. E. Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy, and F. Pfenning, "Trustless grid computing in concert," *Grid Computing - Grid 2002*, vol. 2536, pp. 112–125, 2002.

[11] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky, "Seti@home - massively distributed computing for seti," *Computing in Science & Engineering*, vol. 3, no. 1, pp. 78–83, 2001.

[12] V. Buterin, "The P + epsilon Attack," 2015, accessed 27-May-2015. [Online]. Available: https://blog.ethereum.org/2015/01/28/p-epsilon-attack/

[13] M. Yurkewych, B. N. Levine, and A. L. Rosenberg, "On the cost-ineffectiveness of redundancy in commercial P2P computing," *Proc. ACM Conf. Comput. Commun. Secur., 12th, Alexandria, Va., Nov. 7–11*, pp. 280–288, 2005.

[14] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, accessed 28-May-2015. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[15] V. Buterin, "An exploration of intrinsic value: What it is, why bitcoin doesn't have it, and why bitcoin does have it," 2011, accessed 28-May-2015. [Online]. Available: https://bitcoinmagazine.com/8640/

[16] ——, "Ethereum: A next-generation smart contract and decentralized application platform," 2015, accessed 21-April-2015. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[17] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Addison-Wesley, 2001, pp. 327–328.

[18] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2015, accessed 21-April-2015. [Online]. Available: http://gavwood.com/Paper.pdf

[19] Ethereum, "JSON RPC," 2015, accessed 21-April-2015. [Online]. Available: https://github.com/ethereum/wiki/wiki/JSON-RPC

[20] ——, "JavaScript API," 2015, accessed 21-April-2015. [Online]. Available: https://github.com/ethereum/wiki/wiki/JavaScript-API

[21] ——, "Ethereum Contract ABI," 2015, accessed 05-May-2015. [Online]. Available: https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI

[22] ——, "go-ethereum," 2015, accessed 14-May-2015. [Online]. Available: https://github.com/ethereum/go-ethereum

[23] ——, "cpp-ethereum," 2015, accessed 14-May-2015. [Online]. Available: https://github.com/ethereum/cpp-ethereum

[24] ——, "ethereumj," 2015, accessed 14-May-2015. [Online]. Available: https://github.com/ethereum/ethereumj

[25] ——, "pyethereum," 2015, accessed 14-May-2015. [Online]. Available: https://github.com/ethereum/pyethereum

[26] Docker, Inc., "Understanding Docker," 2015, accessed 27-May-2015. [Online]. Available: https://docs.docker.com/introduction/understanding-docker/

[27] ——, "Docker Security," 2015, accessed 27-May-2015. [Online]. Available: https://docs.docker.com/articles/security

[28] ——, "Dockerfile Reference," 2015, accessed 27-May-2015. [Online]. Available: https://docs.docker.com/v1.6/reference/builder/

[29] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 6th ed. Pearson Education, 2012, pp. 349–352.

[30] ——, *Computer Networking: A Top-Down Approach*, 6th ed. Pearson Education, 2012, p. 352.

[31] M. Boucadair, F. Dupont, R. Penno, and D. Wing, "Universal plug and play (UPnP) internet gateway device – port control protocol interworking function," RFC (Request For Comments), 2013, accessed 14-May-2015. [Online]. Available: https://tools.ietf.org/rfc/rfc6970.txt

[32] Pivotal Labs, "Pivotal tracker: Getting started," 2015, accessed 21-April-2015. [Online]. Available: https://www.pivotaltracker.com/help/gettingstarted

[33] Ethereum, "Alethzero," 2015, accessed 21-April-2015. [Online]. Available: https://github.com/ethereum/cpp-ethereum/wiki/Using-AlethZero

[34] Raspberry Pi Foundation, "Raspberry Pi 2 Model B," 2015, accessed 21-April-2015. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

[35] H. McCook, "An Order-of-Magnitude Estimate of the Relative Sustainability of the Bitcoin Network," 2014, accessed 29-April-2015. [Online]. Available: https://cdn.panteracapital.com/wp-content/uploads/The-Relative-Sustainability-of-the-Bitcoin-Network.pdf