Masterarbeit / Master Thesis

Titel der Masterarbeit / Title of the Master's Thesis

A Blockchain Enabled WS Agreements Framework

Verfasst von / Submitted by

Stefan Starflinger BSc

angestrebter akademischer Grad / in partial fulfillment of the requirements for the degree of

Master of Science
MSc

Wien, 2018/ Vienna, 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet	Masterstudium Wirtschaftsinformatik UG2002
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Abstract

The increase in the supply of on-demand computing resources and the ever-growing number of internet capable devices calls for a new approach in provisioning cloud computing resources. Through trustless multi-round bilateral negotiation, we present an alternative to the current 'off the shelves' approach. Much research has gone into Service Level Agreements (SLAs), from how they are defined, negotiated, and monitored to how they evolve. However, seeing as how trust is a necessity for SLA negotiations, researchers have not focused on ways to remove the need for trust. We demonstrate that an agreement between a consumer and a provider can be reached, without either having to trust the other. We spotlight a method, which lets consumers and providers negotiate for cloud resources. Our focus lies on storing negotiation offers and the resulting agreement in a Blockchain. That way, agreements are tamper proof, and their provenance is evident. This work opens the doors for integrating smart contracts into the negotiation and agreement process. As an example we implemented a referee that can characterize the quality of a service by detecting discrepancies between the terms described in the SLA, and the actual values provided. We found that negotiating over the Blockchain is still expensive, but the additional security implications are promising.

Zusammenfassung

Das steigende Angebot an Computing-Ressourcen und die ständig wachsende Zahl internetfähiger Geräte erfordern eine neue Art der Bereitstellung von Cloud-Ressourcen. Während viele Themen im Bereich "Service Level Agreements" (SLAs) gut erforscht sind, besteht Nachholbedarf im Bereich Vertrauen bei SLA-Verhandlungen. Wir zeigen, dass ein Abkommen zwischen einem Verbraucher und einem Anbieter erzielt werden kann, ohne, dass einer dem anderen vertrauen muss. Dabei stellen wir eine Methode vor, mit der der Verbraucher und der Anbieter eigenständig über Cloud-Ressourcen verhandeln können und die daraus resultierende Vereinbarung dauerhaft in einer Blockchain verewigt wird. Auf diese Weise sind Angebote und Vereinbarungen manipulationsgeschützt und ihre Datenherkunft offenkundig. Diese Arbeit öffnet Türen für die Integration von "Smart Contracts" in die Verhandlungs- und Vereinbarungsprozesse. Als Beispiel haben wir einen Schiedsrichter implementiert, der die Qualität einer Dienstleistung charakterisieren kann. Diskrepanzen zwischen der SLA-Vereinbarung und den tatsächlich gelieferten Leistungen werden verewigt. Unsere Ergebnisse zeigen, dass es immer noch teuer ist Verhandlungen auf einer Blockchain abzubilden. Jedoch macht die erhöhte Sicherheit es zu einem vielversprechenden Modell für die Zukunft.

Contents

List of Figures v

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem . . . 2
- 1.3 Decentralized Trust . . . 3
  - 1.3.1 Distributed Ledger Technologies . . . 3
  - 1.3.2 Blockchain Adoption . . . 4
- 1.4 Contribution and Structure . . . 5

2 State of the Art 7
- 2.1 Distributed Ledger Technologies . . . 7
  - 2.1.1 Overview . . . 7
  - 2.1.2 Bitcoin . . . 9
  - 2.1.3 Ethereum . . . 11
  - 2.1.4 Iota . . . 12
  - 2.1.5 Neo . . . 13
- 2.2 Service Level Agreements . . . 14
  - 2.2.1 Definition . . . 14
  - 2.2.2 Web Services Agreement and Agreement Negotiation . . . 14
  - 2.2.3 Alternatives . . . 15

3 Requirement Analysis 16
- 3.1 Functional . . . 16
  - 3.1.1 Discussion on the choice of Blockchain . . . 16
  - 3.1.2 Smart Contracts and Transactions . . . 17
  - 3.1.3 Multi-Round Negotiation . . . 19
- 3.2 Non-Functional . . . 19
  - 3.2.1 Privacy . . . 19
  - 3.2.2 Availability . . . 20
  - 3.2.3 Auditability . . . 21

4 Specification **22**
4.1 Diagram . . . 22
4.2 Description . . . 23
4.2.1 Initiate Negotiation . . . 23
4.2.2 Store Offers . . . 24
4.2.3 Negotiate . . . 26
4.2.4 Referee . . . 27

5 Technology Stack **29**
5.1 Layers of Adoption . . . 29
5.2 Definitions . . . 30
5.2.1 Verifiable Negotiation . . . 31
5.2.2 Negotiation Contract . . . 31
5.2.3 Resource Address . . . 31
5.3 Negotiation Contract . . . 31
5.3.1 Parameters . . . 32
5.3.2 State transitions . . . 33
5.4 Bilateral Negotiation Model . . . 34
5.5 Implementation . . . 35
5.5.1 Dependencies . . . 36
5.5.2 Decentralized Storage . . . 37
5.5.3 Deployment . . . 38
5.5.4 Application Interface . . . 39
5.5.5 Contract Code . . . 43

6 Use Case **48**
6.1 Simple Negotiation . . . 49
6.2 Negotiation Strategy . . . 51
6.3 Cloud Referee . . . 53

7 Evaluation **54**

8 Future Work and Conclusion **56**
8.1 Future Work . . . 56
8.2 Conclusion . . . 57

Bibliography **58**

List of Figures

2.1 Merkle Tree 12
2.2 Contract Function Call 13

3.1 Transaction State Change 18

4.1 Use Cases 22

5.1 Layers of Adoption 30
5.2 Contract States 34
5.3 Bilateral Negotiation 35
5.4 Write to IPFS 38
5.5 Read from IPFS 38
5.6 Deployment Diagram 39

6.1 Simple Negotiation Example 50
6.2 Negotiation Strategy Example 52
6.3 Referee sets a Flag 53

7.1 Gas Cost 54

1 Introduction

In the following sections we will give an introduction to Service Level Agreements (SLAs) and Distributed Ledger Technologies (DLT). We talk about the motivation of why SLAs and DLTs are interesting and how they can be used together.

1.1 Motivation

The access to computing resources is changing ever more from a capital expenditure model to an operational expenditure model Armbrust et al. (2010). Cloud computing has seen major investments by big cooperations. For example, Microsoft spent $9 Billion on operating their 'Intelligent Cloud' in 2017 and earned an income of $27 Billion in the same period[1]. Apart from the 'Intelligent Cloud' income stream, Microsoft has two other categories of income, making the 'Intelligent Cloud' income about a third of their total revenue[2]. Access to distributed computing resources has been categorized as the fifth utility after electricity Buyya et al. (2009). Although, cloud computing is only one form of distributed computing it has become the most popular. Unfortunately, the current landscape of cloud services forces consumers to obtain their services from a single cloud provider. This is due to the lack of standardization between different cloud providers, which makes it difficult to switch, a phenomenon known as vendor lock-in Opara-Martins et al. (2014). It takes much consideration and planning to avoid vendor lock-in when building a solution on top of a cloud service. To improve the service landscape, a marketplace needs to be formed that allows cloud providers to compete, moving away from the so-called 'off the shelves' approach Pittl et al. (2018). At the current level of standardization, the condition under which a competitive market can be formed for healthy competition is not given. In a sense, there is a need for more market liquidity that would allow providers to compete.

The introduction of standards would mean that providers could offer their services in a more competitive market. In a more competitive market cloud providers are more incentivized to

[1] https://www.microsoft.com/en-us/Investor/earnings/ [01.09.18]
[2] https://view.officeapps.live.com/op/view.aspx?src=https://c.s-microsoft.com/en-us/CMSFiles/FinancialStatementFY18Q4.xlsx?version=10864f3a-8c49-ee89-6d6b-c9af3ee53d1b [01.09.18]

follow standards Baig et al. (2017); Dastjerdi u. Buyya (2015). Amazon has made first steps in the direction of creating a competitive market with their Amazon Spot[3] market. The next step would be to open the spot market up for other providers, i.e., create a similar solution where different providers can participate. An example of such a market place would be the attempt of the Deutsche Boerse. They tried to create a cloud market place[4], but failed as they could not generate enough demand[5].

1.2 Problem

Across the board we see companies, no matter the business model or the size, migrating their services to the cloud Buyya et al. (2009). Not having to buy hardware and develop back-end systems is a huge cost saver, but does have its drawbacks described by Takabi et al. (2010). In the cloud, companies are no longer in charge infrastructure. They give the responsibility away to their providers. If the provider does not deliver companies could see themselves losing customers, money or worse. Another downside of cloud computing is trust, which is the probability that everything will go right[6]. Unfortunately, the black box nature of cloud computing does not provide the consumer with the necessary information to determine the probability mentioned above. The next best option that consumers have is to monitor the resources that they receive, which is difficult and expensive, so most consumers don't end up doing it. To reduce the dependency on trust Service Level Agreements (SLAs) are necessary to define QoS attributes and recovery procedures.

It is possible to compare SLAs to insurances contracts, they don't prevent unforeseen events and failures, but they ensure that the customer is covered if something were to happen. SLAs help manage risk and reduce the dependency on trust. In the unlikely event of an outage responsibilities and actions need to be clearly defined. The consumer has a different goal compared to the provider. The provider and the consumer both have a utility function. If the utility functions do not cross an agreement cannot be made. To make it possible to adjust expectations i.e. adjust the utility function the negotiation partners have to exchange multiple offers and counteroffers. For that reason single round negotiation, as described in Andrieux et al. (2007), is insufficient. Bilateral multi-round negotiation, as described in Waeldrich et al. (2011), is necessary. Customers that inquire about SLAs receive Guarantee Terms that diverge considerably.

[3] https://aws.amazon.com/ec2/spot/
[4] https://web.archive.org/web/20151001060015/https://cloud.exchange/ [16.09.2018]
[5] https://www.speicherguide.de/news/deutsche-boerse-cloud-exchange-schliesst-ihre-wolkigen-pforten-21873.aspx [16.09.2018]
[6] https://www.ted.com/talks/rachel_botsman_we_ve_stopped_trusting_institutions_and_started_trusting_strangers

Most big cloud providers (Amazon, Microsoft and Google) offer their services on a take it or leave it basis. Negotiating SLAs is not very common, but would leave the consumer and the provider better off. Resources, which consumers cannot afford are left idle while providers still have to pay operating expenses. A market like system would account for fluctuations in supply and demand. No providers that are known to us allow for multi-round bilateral negotiation of SLA offers and agreements. Moreover, in Dastjerdi u. Buyya (2015) paper he argues that to improve the overall cloud market utility, providers need to adopt standards to optimize resource utilization, negotiate autonomously, and benefit from market forces.

1.3 Decentralized Trust

Individuals and companies tend to stick with a provider whom they can trust. Amazons reputation helps them acquire customers, and the massive switching cost helps them retains customers. The authors Ranaweera u. Prabhu (2003a, b) found that satisfaction and trust are two reliable indicators of customer retention. They mention that high switching costs can retain even dissatisfied customers. Considering an extract from the sharing economy Hamari et al. (2016), consumers tend to trust providers more if they are capable of delivering what they advertise. Trust is lost if the provider fails to deliver. Amazon is well established with a good track record where customers can be sure of what they will get.

Trust is a non-functional attribute that is considered when humans make decisions, yet autonomous negotiation algorithms cannot make decisions based on trust without being able to quantify it. A means of negotiating verifiable SLA is needed, which can be obtained through integrating a distributed ledger into the negotiation process. By going trustless and adding standardization we take away a lot of the switching costs and are giving more of the welfare back to the consumer. Further, Feng et al. (2014) concluded that cloud provider that do not focus on capacity but that focus heavily on SLA to deliver QoS attributes are just as competitive even if they are five times smaller.

1.3.1 Distributed Ledger Technologies

An example of a Distributed Ledger Technology (DLT) is a Blockchain. Nakamoto (2008) introduced the concept of a Blockchain following the global financial crisis. Today, it could help shape the future of a borderless financial system. It eradicates the need for a trusted third party. Instead, we lay our trust in a public, verifiable protocol. The Blockchain protocol combines a reward system with a consensus algorithm in a novel way that keeps the system secure and

stable. In his/her paper Nakamoto (2008) introduced a new currency named Bitcoin, which would later become the first implementation of a Blockchain. The Bitcoin paper describes a smart recombination of known concepts that allow users to reach consensus over the internet. It allowed for the creation of electronic cash built on a solid mathematical foundation. The solution consists of a chain of blocks, known as a Blockchain, where each block contains a set of transactions. The addition of a poof of work algorithm allows for new blocks to be added to the chain. Each block references the previous block, and only the longest chain is valid. The umbrella term under which Blockchain Technology is known is defined as Distributed Ledger Technology and by now encompasses thousands of different Blockchain and Blockchain like implementations. Some of the most prominent implementations according to their market capital are Bitcoin, Ethereum and Ripple[7].

1.3.2 Blockchain Adoption

In the WS-Negotiation Framework, participants, namely the negotiation initiator and responder, exchange information about their offers. Each participant is responsible for storing and managing the transferred documents. There are no standards in place to provide consistency between the exchanged offers. If a participant modifies an offer, the other participant has no idea, and disputes could occur. It would be the provider's word against the consumers. If the initiator and responder were to share a database, then both would need write access. Both could modify the shared database and any entry in it. That is why there is a need for a shared database secured through DLT. Both participants can write to the database without requiring faith in the other participantâs trustworthiness If the participants trust each other then no Blockchain is necessary, but there are many providers and consumers, and the chances of needing resources from an unknown source is high. The Blockchain offers a form of insurance that offers little room to dispute what has been agreed upon. So the question that a consumer or provider should ask is:

```
Do I trust my negotiation partner?
```

If the answer is *yes*, then it is not necessary for you to use a Blockchain. Yet if the answer is *maybe* or *no* then a Blockchain might be a good idea. Below we look at what kind of Blockchain might be suitable for bilateral negotiation and how the individuals would benefit from using a Blockchain instead of a regular database or no database. It might be difficult to judge if your trust is misplaced or not, in that case, be sure to know what the risks are. If the risks are too high, it might be better to use a Blockchain even if you trust your negotiation partner.

[7] as seen on https://coinmarketcap.com/ 28.08.2018

1.4 Contribution and Structure

In the paper Pittl et al. (2018) introduces a solution to the 'off the shelves' cloud market problem by developing a Bazaar-Blockchain. They discuss two research questions, the first is to establish a Blockchain for the negotiation process, and the second is to use smart contracts for the execution of the resulting agreements. We tackle these research question as well with a slightly different approach. Instead of only using smart contracts for the agreements we use them for both the negotiation process and for the resulting agreement. Further, we will not be using a custom built Blockchain solution, but instead benefit from an existing implementation namely Ethereum Wood (2017). We present a framework of best practices that allow for the use of the WS-Agreement Negotiation standard with the Ethereum Blockchain. Further, we integrated the interplanetary file system (IPFS) Benet (2014) to allow for off chain decentralized storage of the WS-Agreements.

Below we will describe the structure of the paper and the contributions that are made in each chapter. We will look at SLAs, what they are and why they are essential. There are five service deployment phases as described by Dastjerdi u. Buyya (2012, 2015). The phase we are going to concentrate on is the SLA Negotiation phase. The scaling and decommissioning phases are not considered. Next, we look at how the 'off the shelves' approach of the current SLA landscape can be improved, by introducing bilateral negotiation using the WS Agreement Negotiation Framework. Third, we look at trust in conjunction with Blockchain Technologies. We then talk about our framework and what it can do by providing information on example implementations. After, we discuss the importance of the framework and identify if it is actually useful.

State of the Art

In this chapter, we look at the current state of Blockchain Technologies and of Service Level Agreements. We find that the field of Distributed Ledger Technologies although very new is growing quickly with many different projects and implementations available today.

The contributions of this chapter include

- Introduce the field of Distributed Ledger Technologies (DLT).
- Give a detailed account of different DLT.
- Get an understanding of what Service Level Agreements are.
- Look into the WS-Agreement and WS-Agreement Negotiation Standard.

Requirement Analysis

In this chapter, we will look into the requirements that our framework needs to fulfill. We discuss these requirements and also justify our choice of technology, partly in this chapter, but in more detail in the following chapter.

Specification

To get a better high-level understanding of what our framework does from a more business point of view we created a use case diagram and use case descriptions. The use cases we will discuss are Initiate Negotiation, Securely Store Offers, Negotiate and Monitor Service. Together they make up our framework.

Technology Stack

In this chapter, we compare technologies and chose the best-suited software and hardware for our framework. We describe our implementation and the dependencies used.

The contributions of this chapter include:

- Contract escrow with a dispute function
- Decentralized neogitation without the need for third parties

Use Case

In this chapter, we describe a scenario and go through a concrete example of that scenario using our framework. The scenario will describe the high-level overview and the practical example goes into more detail.

Evaluation

In this chapter, we discuss the results of our work. The benefits of the blockchain come with some downsides. We look at the costs of using a Blockchain.

Future Work and Conclusion

Finally, we talk about open questions and we conclude the paper.

2 State of the Art

The research field of Smart Contracts and Blockchains is still growing rapidly and there are many directions that researchers are exploring. In this chapter, we look into different Distributed Ledger Technologies (DLT) that are under development. The field of Service Level Agreements (SLAs) is more thouroughly researched. We give a short account of what SLAs are and how they are defined. After gaining an understanding of SLAs we look at the WS-Agreement and the WS-Agreement Negotiation standards.

In this chapter we:

- Introduce the DLT research field.
- Give a detailed account of different DLT.
- Get an understanding of what Service Level Agreements are.
- Look into the WS-Agreement and WS-Agreement Negotiation Standard.

2.1 Distributed Ledger Technologies

With the introduction of Bitcoin the field of DLT has been growing rapidly. There are now thousands of DLT implementations, some of them will be discussed below. The term distributed ledger mainly refers to a group of nodes sharing a database, or in other words, a ledger. Many Uses Cases can be realized with DLTs, one of the most promising is creating an electronic cash system. Previous attempts at creating an electronic cash system failed as they could not solve the double spend problem. The primary research challenges of DLTs are i) usability, ii) scalability and iii) consensus.

2.1.1 Overview

A Blockchain is the most prominent example of a DLT where nodes follow a protocol that is cryptographically secured. Each node has a copy of the ledger, validates all of the previous transactions, and executes the same steps as all the other nodes. The method with which new

blocks get added to the ledger depends on the consensus algorithm. Many different consensus algorithms exist, such as Proof of Work (PoW) introduced by Back et al. (2007), Proof of Authority (PoA) used by private Blockchains, and Proof of Stake (PoS) as presented in Vitalik u. Virgil (2017). Expectations for DLTs are very high. According to Gartner[8] Blockchain has passed the peak of expectation and is currently falling to the Trough of Disillusionment with an estimation of five to ten years to reach the Plateau of Productivity.

The internet is the first big success story of a distributed technology. What the internet has done to the flow of information, the Blockchain will do to the flow of value. Trust will shift from third parties to public protocols. If a group of people does not agree with the maintainers of a Blockchain protocol, they can fork the Blockchain and modify the rule set. They cannot, however, falsify past information in the chain without redoing the PoW. Bitcoin Cash is an example of such a fork[9] from the original Bitcoin Blockchain. They modified the rule set by increasing the block size among other things. This modification came about since people saw that the costs for adding a transaction to the Blockchain were unreasonably high. The high transactions costs caused people to get together and create a fork with a rule set that increased the number of transactions that a block could carry and thus reducing the competition for block space. The fork uses the same underlying consensus algorithm and the same PoW algorithm making it a direct competitor. When choosing a chain, it is important to consider the tradeoff between security and transaction cost. The hash power of Bitcoin could rewrite the Bitcoin Cash chain in under a month[10].

Trust

Since the beginning of time people have been trading goods. In the beginning, people exchanged goods for other goods. This simple form of a transaction was feasible in small societies but as society grew a more efficient means of trading was necessary. The introduction of fiat currency was a solution that allowed for someone to easily exchange goods for coins. With these coins, the person could then buy another good. It was important for those coins to be hard to fake otherwise someone could purchase goods without actually providing any value to society. Further, the entity that issued the money needed to be trustworthy. If the issuer is not trustworthy, then they could devalue the currency. We trust institutions not to be corrupt and have our interest at heart. Unfortunately, there have been many examples in the past where this trust was broken. With the introduction of Blockchain technologies, it has become possible to distribute value without the need to trust an institution.

[8] https://gartner.com
[9] https://bitcoin.com [30.08.18]
[10] fork.lol, http://bitcoin.sipa.be/index.html [30.08.18]

The reason trust is no longer necessary is that the guarantees that the creators make can be verified. In the case of Bitcoin, an example of a guarantee is that only the holder of a public-private key pair can spend the coins that are associated with their public key. Further, all of the code is open-source, and anyone can run a node. This means that, given enough time, anyone can understand the protocol and validate that the Blockchain does what the creators advertised. Having a shared database means that everything is verifiable and public. For everyone to have access to the database is problematic since people do not want to have their financial information out there. The solution Nakamoto (2008) mentions is not a very good solution. Satoshi Nakamoto says that as long as people cannot be associated with their public address, they should be fine. Most Blockchain solutions have pseudonymous addresses. As soon as people get associated with their address, other people can be inferred from the transactions. The possibility to infer information means that future revelations of identity can be disastrous. If someone knows your address and your identity, he or she can derive, from the public ledger, all of your transactions. That way they know how much you are worth and to whom you have sent money, which has huge privacy implications as analyzed by Androulaki et al. (2012). It is possible to create a new key pair every time someone receives money, but this quickly becomes difficult to manage. Given these advantages and disadvantages, we now investigate whether a Blockchain is even necessary for our use case.

2.1.2 Bitcoin

Satoshi Nakamoto first proposed the idea for a peer to peer cash system named Bitcoin in his paper Nakamoto (2008). There he described an electronic cash system that does not need a trusted third party to perform payments. Together with a group of developers, Satoshi Nakamoto set out to bring his idea to life. He drew his motivation from the 2008 financial crisis, where banks brought ruin over the global financial markets. The fundamental idea behind Bitcoin is now known as the Blockchain, which has the potential to revolutionize our industry.

Double Spending

The major crux, of realizing an electronic cash system, is the double spending problem. It sums up to ensuring that when Alice receives a payment from Bob, how can she be sure that Bob has not already spent that money somewhere else. That is where the distributed ledger technology (DLT) comes into play. It is a ledger equipped with cryptographic proofs. To prevent double spending, the ledger is secured through a mechanism called proof of work similar to the one used by Back et al. (2007).

Proof of Work

The cornerstone of why Bitcoin works is its Proof of Work (PoW) algorithm. It rewards honest nodes in the peer to peer network and simultaneously punishes dishonest nodes. This form of incentive is only the case so long the honest nodes control the majority of the hashing power. The nodes that secure the network (the honest nodes) can receive a reward for their work. The work that needs to be done, in the form of hashing operations, serves to reach consensus and keep the network secure. Honest nodes that secure the network are also known as miners. These miners perform hash operations and the first miner to calculate a hash lower than the difficulty is rewarded by the protocol with a predetermined amount of Bitcoin. Further, the algorithm used for mining is the $sha256^2$, which can also be represented as $sha256(sha256(X))$. Miners adjust the nonce of a block header repeatedly to find a solution and are then allowed to create a new block and attach it to the chain. As an analogy, imagine the Blockchain as a puzzle, where each piece only has two sides. A new piece can only be placed at the end of the current chain. The images depicted on each puzzle piece are the transactions in a block. The miners, as quickly as possible, take out pieces from a huge bag. To be more precise, the bag that the miners choose puzzle pieces from, there are around $2^{256} = 1.158 * 10^{77}$ possible combinations. Only a small subset of these combinations determined by the difficulty are eligible for a reward. They check if the piece fits if it does they pass a copy of that puzzle piece to all other nodes and get a reward from the protocol. Although, the other miners only accept the piece if the image fits the previous pieces i.e. if the transactions in the new block are not already spent in a previous block.

The double spending problem is prevented by using the proof of work algorithm and waiting for enough other nodes to confirm the chain by attatching a new block. Only the longest chain of blocks is the valid chain and the other blocks that do not make it onto the longest chain are called uncles. A retail seller would be advised to wait for multiple blocks to be attached (confirmations) to the the longest chain before shipping an article.

Privacy

Banks keep people's information and transactions confidential, but a public distributed ledger cannot do the same. Nakamoto argues that the pseudonymous addresses grant the user privacy in the public Blockchain. To ensure that someoneâs identity does not get discovered they should create a new key pair for each transaction. A paper by Androulaki et al. (2013) discovered, in a university setting, that even when adopting these privacy measures 40% of participants

were identified. Since Bitcoins genesis, other developers have tried to improve on the privacy aspect.

Fungibility

With traditional cash it is nearly impossible to tell who the previous owner was. It is also not possible to tell for what a cash bill was used. Someone could have bought drugs or a fruit with the money and it would make no difference to the person now in posession of the cash bill. This is known as Fungibility, which is a given for cash but it is not a given for Blockchains.

Addresses used for purchasing illegal goods can be identified through public forums or similar means. The Bitcoin used in an illegal transaction will forever be tainted. Addresses that have been identified as having been used for malicious activity will contain Unspent Transaction Outputs (UTXO) i.e. Bitcoins that are less desirable. These addresses can be monitored by law enforcement.

2.1.3 Ethereum

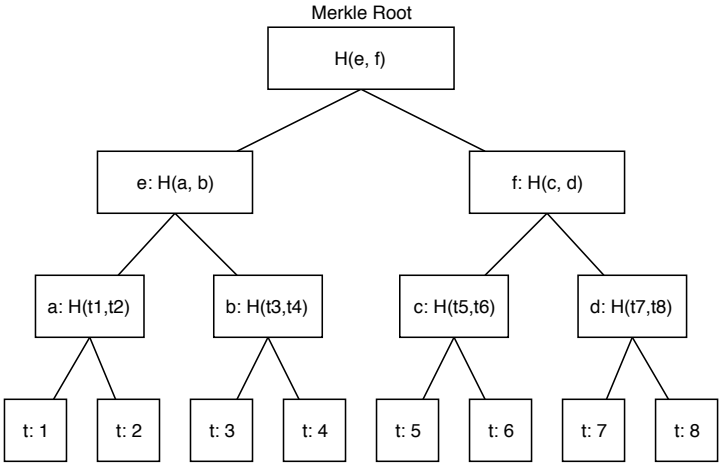
Ethereum was first proposed by Vitalik Buterin in 2013 and later described as an alternative to Bitcoin in his whitepaper [11]. It uses the same underlying Blockchain Technology as Bitcoin but tries to improve certain aspects. The general idea was to create a Turing-complete language that can be run on the Blockchain. This idea was novel compared to the simple scripting language that Bitcoin implements. Through the Turing complete language, Buterin gave developers a playground on which they could develop their trust-less decentralized ideas. These are currently called Decentralized Applications or DApps short.

Ethereum allows users to write smart contracts using its Turing-complete solidity programming language. The code is compiled down and run on a virtual machine, similar to how Java runs in the JVM. This VM is called the Ethereum Virtual Machine (EVM). The specification for the EVM is described by Wood (2017) and the reference implementation was done in the programing language go[12]. Many different clients have since been created in different programming languages. This further increases the amount that Ethereum is decentralized as the development is no longer dependent on one programming language and one group of supporters.

[11] The paper is hosted on GitHub where anyone can create a pull request to improve or translate it https://github.com/ethereum/wiki/wiki/White-Paper [06.05.18]
[12] Github repository of the reference implementation https://github.com/ethereum/go-ethereum [06.05.18]

Merkle Trees

Figure 2.1: Example of a Merkle Tree where t stands for a transaction and H is a hash function. The notation a: H(t1, t2) means that a will be assigned with the resulting hash value of hashing transaction 2 and transaction 2.



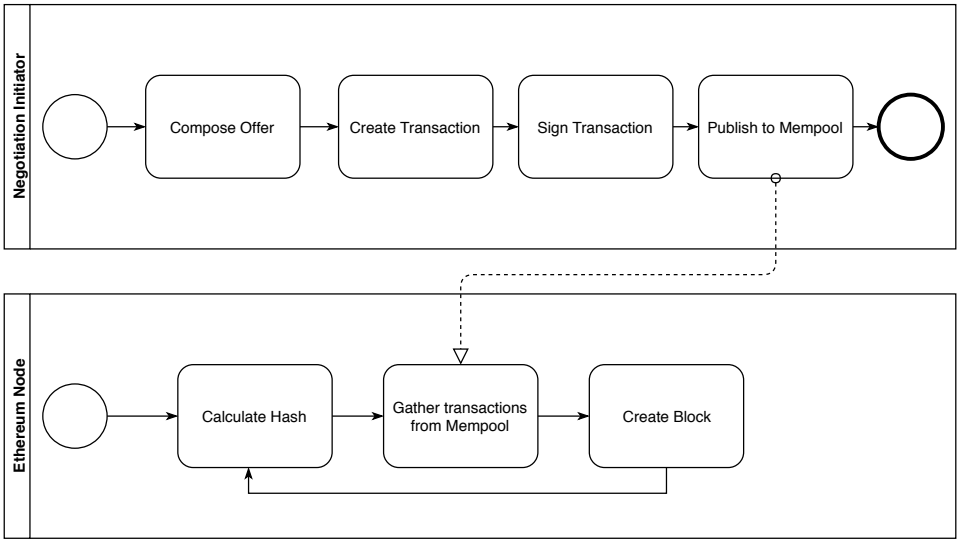
Merkle trees Merkle (1980) were added to reduce the amount of information a node needs to store while still being able to validate new blocks. In Figure 2.1 we can see how a Merkle tree is built. Transactions are paired and hashed. The resulting hash is paired with another hash that was derived in the same way. Continuing the procedure for all transactions creates a tree that is built from the bottom up. The resulting root node is called the Merkle root and contains a hash value that represents the previous nodes beneath it. Any changes made to a node in the tree would result in a different Merkle root value. After some time nodes could prune the Blockchain and delete transactions to free up space without compromising on security.

The Figure 2.2 shows a simplified version of the process of interacting with a smart contract i.e. creating transactions that get mined and added to the Blockchain. Although the only function calls that change the state of the Blockchain will need to be mined. Other function calls do not need to be mined. These function calls are the ones that do not change the state of the Blockchain but rather just retrieve information from the chain.

2.1.4 Iota

In his whitepaper Serguei (2018) describes problems with the Bitcoin protocol and he presents an alternative solution, Iota. Bitcoin being the first implementation of a distributed ledger lead to people discovering many disadvantages with the protocol. Iota tries to tackle these

Figure 2.2: The processes that take place when a contract state is modified by creating an offer



flaws, keeping the internet of things (IoT) in mind. For example, with the Bitcoin protocol, microtransactions are not economically viable, since the transaction fee is too high.

One of the critical points with the Iota distributed ledger is that it does not have transaction fees. Instead, before broadcasting a transaction, peers need to secure the network by performing a proof of work for two previous transactions. In doing so, Iota does not split the network into two groups, the miners, and the users. Another critical difference is that, instead of having a single chain, Iota adopted a graph approach, a Directed Acyclic Graph (DAG) to be precise. To propagate a transaction, the sender has to validate two previous transactions. That way the network can grow and become more secure the bigger it gets. The downside is that it is more easily attacked while the network is still small. For that purpose, there is a central coordinator that secures the network while it has not reached a critical mass of transactions Bramas (2018).

2.1.5 Neo

Originally branded AntShares, NEO was first released in February of 2014[13]. The goal of the project is to create a smart economy. This smart economy would be constructed on digitized physical assets exchanged through smart contracts. Additionally, NEO offers secure digital identities in accordance with the Public Key Infrastructure X.509 standard. While providing anonymity, NEO therefore provides a viable service for entities whose actions must comply with

[13] Neo Whitepaper can be found at http://docs.neo.org/en-us/ [06.05.18]

financial regulations. It supports a wide range of common programming languages making it accessible to existing industry professionals and decreasing the resources required for retraining staff. Similarly to Ethereum, NEO supports the development of decentralized applications. It employs a delegated Byzantine Fault Tolerance (dBFT). The consensus algorithm of Neo is not based on the Proof of Work as in Ethereum and Bitcoin, but rather Proof of Stake (PoS) algorithm. This means that the security of the system is given by people staking their funds instead of providing compute capacity.

2.2 Service Level Agreements

we start off with the definition of Service Level Agreements (SLAs) and then go into more detail about why they are important and how they are defined. Especially the standards that have been established to negotiate SLAs will be looked at.

2.2.1 Definition

Service Level Agreements are legally binding agreements about the service that is provided to a consumer. Similarly Haq (2010) defined it as:

> A Service Level Agreement is a formal, legal contract between a service provider and a consumer that specifies, in quantifiable terms, what service level guarantees the service provider will deliver, and it defines the consequences (penalties) if the service provider fails to follow through with said commitments.

To this definition, there are two main parts. The first is that the extent of the commitment that the provider is willing to offer to the consumer must be laid out. The second is the punishment that is due if the provider is not able to uphold his commitment. Before a SLA is binding, it needs to go through a negotiation process, which will be looked at more closely below and in a later section 3.1.1. For now, it is enough to realize that it is an agreement between two parties over a range of attributes that a service must provide.

2.2.2 Web Services Agreement and Agreement Negotiation

The Web Services Agreement Specification Andrieux et al. (2007) offers a normative method for exchanging SLA offers and agreements. It was created by the Open Grid Forum and uses XML as a means to describe and exchange offers. Further, operations are defined for managing the SLA life-cycle. The basic communication flow does not allow negotiation, but it allows for

an agreement to be rejected or accepted. So an offer that is created from a set of templates can either be accepted or rejected by the provider, with no previous context. The lack of negotiation context is why the standard was extended by Waeldrich et al. (2011). The extension introduced multi-round negotiation capabilities, while still being compliant with the Web Service Agreement specification Battré et al. (2010). A resource provider can now with the extension send a counter offer if the terms that the consumer presents are unreasonable. Effectively, a negotiation initiator can request and analyze the agreement templates from a negotiation responder, and later advertise to the negotiation responder, what they are interested in. The negotiation responder can then start a negotiation process with the negotiation initiator.

The structure of an offer is as follows: it starts with a *Name* to identify the offer then a *Context* to describe the participating parties. Information such as the initiator, responder and expiration time can be found here. Next, there are the *Terms* that can be further divided into *Service Terms* and *Guarantee Terms*. The Service Terms describe the quantifiable aspects such as CPU and RAM. The Guarantee Terms on the other and describe the Terms that are more difficult to quantify such as availability. The Guarantee Terms are to be considered in combination with the Service Terms as they describe the Quality of Service (QoS) that a provider should provide for the given Service Terms.

The Service Level Objectives (SLO) are bounds in which a service should be provided. Are assurances that can come in the form of constraints in the offers. For example, an offer from a consumer to a provider can contain a constraint that the incident response time needs to be between 3 and 5 hours.

2.2.3 Alternatives

The European Commission published a report written by Blasi et al. (2013) on research projects. The main SLA standard mentioned in the report was the WS-Agreement standard. A possible competitor for the WS-Agreement standard is the Web Service Level Agreement Language (WSLA) by Nepal et al. (2008), but IBM - the creators of the standard - joined the Open Grid Forum as stated by Haq (2010). Another alternative is the SLAng language Lamanna et al. (2003), but that language does not support negotiation.

3 Requirement Analysis

We look into the requirements that our framework needs to fulfill and justify our choice of technology, partly in this chapter, but in more detail in the following chapter.

3.1 Functional

In this section we discuss the functional requirements that our framework needs to fulfill.

3.1.1 Discussion on the choice of Blockchain

In this section, we look at the defining factors of the Blockchain solutions discussed in the previous chapter, to decide which one to use. We look at the advantages and disadvantages and then come to a conclusion. An overview can be seen in the table 3.1 where different parameters of the Blockchain Technologies are compared. The first solution we looked at is Bitcoin. Bitcoin offers a scripting language that allows for minimum extensibility of the core transaction capabilities Seijas et al. (2016). The language is not Turing-complete, and the lack of state manipulation makes it difficult realizing specific use cases other than transferring coins. Bitcoin was created with particular transactions in mind, namely changing the state of UTXO. Agents that control funds, such as multi-signature contracts can be realized.

Name	**Type**	**Genesis**	**Consensus**	**Block Time**
`Bitcoin`	Blockchain	2009	PoW	600s
`Ethereum`	Blockchain	2014	PoW	15s
`Neo`	Blockchain	2016	PoS	20s
`Iota`	DAG	2015	adjusted PoW	variable

Table 3.1: A table comparing different Distributed Ledger Technologies (DLT). The table compares key factors of the different DLTs. The first is the name then the type of DLT. The third collumn is the year in which the genesis block was mined. Next, is the type of consensus algorithm used by the Blockchain. Finally, the block time describes the time between block creation.

After Bitcoin, we looked at Ethereum which offered a more flexible means of changing state in the Blockchain by integrating a Turing-complete programming language. The language allows developers to realize any Use Case where trustless state management is critical on top of an existing Blockchain. The next technology we considered is Iota, which is not a Blockchain in the classical sense. It focuses on IoT devices and has a modified PoW algorithm that is only performed when a transaction is propagated. Unfortunately, Iota is not yet usable as it is still very immature. It has not reached the critical mass of transactions to be secure and decentralized without the coordinator. The developers have only recently released a project called Qubic[14] that allows for the development of Smart Contracts. The final project from China is Neo. It has a different consensus algorithm (PoS), which makes it more environmentally friendly. It also has Turing-complete Smart Contracts, which can be programmed in existing languages such as C#. Neo would be a great option, but there is not enough research done in the direction of PoS BitFury (2015). Much research has been conducted about Byzantine Fault Tolerance that Neo uses, but not in the context of a Blockchain Vitalik u. Virgil (2017). Finally, the level of decentralization of Ethereum is higher than in any of the other DLTs. There are multiple reference implementations of the Ethereum client which makes the protocol independent of one group of developers. The other DLTs so far mainly have one reference implementation[15]. For all those reasons we believe that Ethereum is the most solid implementation of a DLT currently available. This could quickly change in the future as the space is still evolving.

For the implementation of our framework, we chose Ethereum. It offers a base layer on which almost any Blockchain domain-specific use case can be realized. Further, creating a Blockchain for an individual use case without tackling problems that Blockchains currently face, such as scalability, would probably result in an inferior and insecure Blockchain. The Blockchain Technology has brought a shift to development, from making products to making protocols. It has shifted many projects from being proprietary to being open source. This means, that developing a Blockchain, for a specific use case, is not enough. It is necessary to build a community around your solution. If people do not see the value that is provided by a new Blockchain protocol, the protocol will not be used and have no value. Below we look at a table of existing solutions and compare them to find a fit.

3.1.2 Smart Contracts and Transactions

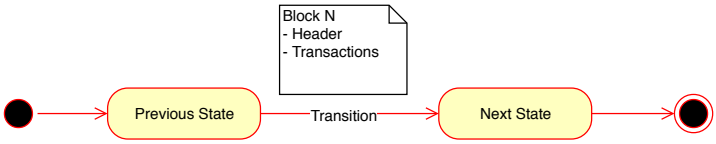
What Ethereum does differently, is abstract the Bitcoin transaction model a step further. Each transaction in Bitcoin is a change in the state of the ledger. The previous state is the

[14] Announced only last week https://qubic.iota.org/ [06.05.18]

[15] There are many implementations the two most promising are Geth and Parity: https://www.parity.io/, https://geth.ethereum.org/, http://ethdocs.org/en/latest/ethereum-clients/choosing-a-client.html

block before, the transaction in the new block changes the state of UTXO. Ethereum adds the possibility to modify state transitions from the current state to the new state through contract code, during the block validation phase. Each node executes Smart Contract code when validating the block.

Figure 3.1: A state transition of the Ethereum Blockchain



Ethereum is a trustless state machine that can execute arbitrary state changes. Smart contracts dictate the state change depending on the input. Below we have a transition that occurs when validating a transaction as seen in Figure 3.1. A transaction, such as a contract call, $contract.ChangeState(input)$ is executed against the $currentState$ which results in a $newState$.

More specifically, a transaction is equivalent to a function call, calling a smart contract causes the state of the Blockchain to transition to a new state. To prevent spamming, each transaction costs gas. The gas is consumed in an infinite loop until the contract returns. To prevent losing all of your funds, it is possible to add a gas limit to a transaction. This limit terminates execution when the gas limit is reached. Creating a contract also costs gas, but that cost only incurs once. As soon as a contract is deployed, functions can be called on that Smart Contract. As such, functions can be written to determine how state transitions occur for inputs that are yet unknown.

The procedure of executing a function call can be seen in Figure 2.2. We have a negotiation participant call a function. The call is in the form of a transaction that needs to be signed by the participant before it is broadcast to nodes. A node saves the broadcast transactions in its Mempool. If the node finds a new block, it will take some of the transactions in its Mempool and add them to the newly created block. Depending on the gas price of the transaction miners can choose to add the transaction to the new block. Miners want to optimize their profit and as such will prioritize transactions that have a high gas price. Miners spend money on hardware and electricity to secure the network. It would not be in their interest to add transactions with a small gas price as they would not be able to pay their bills.

3.1.3 Multi-Round Negotiation

With the Andrieux et al. (2007) it was only possible for a simple negotiation to occur. This causes the offers from different providers to diverge as they do not have sufficient information to adjust their offer. It is a guessing game as the providers do not know what offers the consumer already has. With the extension of the WS-Agreement specification through the addition of multi-round negotiation Waeldrich et al. (2011) more informed offers and agreements can be formed. For example, i) a consumer can initiate bilateral multi-round negotiations with multiple providers then ii) these providers each send offers to the consumer. iii) the consumer can compare the offer with the offers of other providers. iv) After finding the best offer the consumer can send that offer to all other providers to see if they are willing to change their offer. The provider gets more information through the multiple rounds not only on the utility function of the customer but also gets a better understanding of the market.

3.2 Non-Functional

There are many attack vectors from outside as well as from the inside. We focused more on attack vectors between the negotiation participants. Especially integrity and provenance are two key aspects we consider.

3.2.1 Privacy

It is difficult to provide privacy for a Blockchain as the Ethereum Blockchain saves all of the data publicly. However, since we do not store the full offer an agreement on the Blockchain, but rather only store the IPFS hash it is possible to encrypt the IPFS hash with the public key of the negotiation partner. Let $PUBK$ be a public key and PK be a private key. Both the consumer and the provider have a key pair noted as $PUBK_c$, PK_c, $PUBK_p$ and PK_p. If the consumer were to create an offer they would modify the IPFS hash value that points to the document by doing:

$$HASH_{encrypted} = PUBK_p(HASH_{IPFS})$$

When the provider wants to read the offer he would get the $HASH_{encrypted}$ from the contract and decrypt it:

$$HASH_{IPFS} = PK_p(HASH_{encrypted})$$

With encryption only Service Terms that are stored publicly such as variables in the contract such as the duration a contract is valid. Further, both the provider and the consumer do have pseudonymous addresses that make it difficult for anyone to track the negotiations.

Public Ledger

The public nature of the Ethereum Blockchain while still providing some anonymity and allows negotiation participants to make a more informed decision depending on previous agreements. As mentioned everything on the Blockchain is public, and every function can be seen and called by everyone. Preventing unauthorized calls is essential. Modifiers can be defined to prevent unauthorized execution. They are defined using the required keyword and are similar to an assertion. If the assertions are valid, the remainder of the function body gets executed. If the condition defined in the required statement is not met then the call is reversed and the remaining gas that is still available will be returned to the sender. Similarly if the sender tries to transfer too many funds, such a required statement can reverse the transaction and return the remaining Ethereum. The second step after all of the conditions are checked then modifications of the contract can be performed. In the final steps of a function body, other contracts should be called. This sequence is essential to avoid side effects. It is possible to create modifiers that can be added to the definition of a function. These modifiers make it easier to check conditions by adding them to the function definition instead of the function body. Modifiers are not limited to checking conditions but are useful for reducing code verbosity of frequent checks.

3.2.2 Availability

Hosting a Webserver comes with a wide variety of security implications which are not considered in this paper. They are not unique to this use case so other papers will provide more detailed information. We focused only on the integrity and security of the two negotiation partners and not directly on the effects of outside malicious actors. However, with regards to the Blockchain availability is guaranteed through the decentralized nature of the system design.

3.2.3 Auditability

The unique design of the Blockchain allows for state changes to be monitored and viewed by any node in the network. Each state transition gets stored and can be viewed by anyone that has a synchronised chain. Running a node does come with some costs that need to be considered and there are implementations of light and full nodes that vary the amount of data that is stored. The concept of light nodes was enabled through Merkel trees.

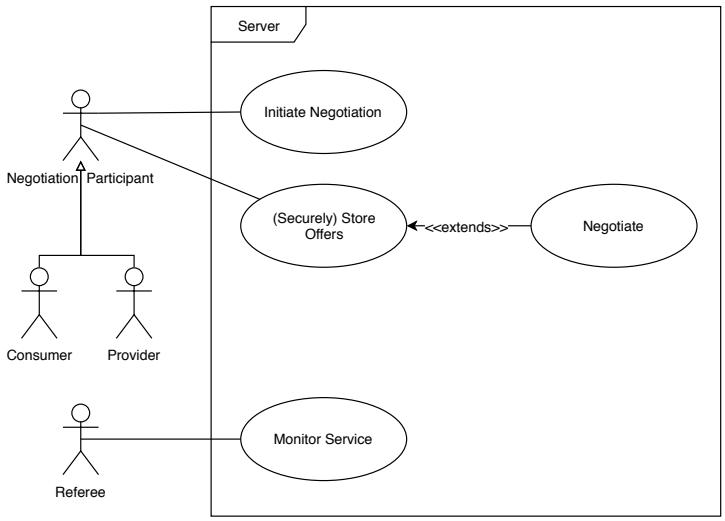
4 Specification

In this chapter, we discuss a high-level overview of what functionality our framework covers and the actors that are involved. The use cases we discuss are Initiate Negotiation, Securely Store Offers, Negotiate and Monitor Service. Together they make up our framework.

4.1 Diagram

The use case diagram in Figure 4.1 gives a high-level overview. Our framework runs on a server that covers four distinct use cases. The actors involved are Negotiation Participants and Referees. The Negotiation Participants can be further sub-divided into consumers and providers. A Negotiation Participant has the possibility to store an offer in a negotiation participate securely. Further, a Negotiation Participant can use the Negotiate extension to run their negotiation strategy over our framework securely.

Figure 4.1: This use case diagram shows the different actors that are involved as well as the individual high level use cases that our framework covers.



4.2 Description

In this section, we discuss each use case in more detail. We list the primary actor that is responsible for the use case. In some cases, we list the stakeholders and what their interest is in the use case. Next, we describe the use case followed by an example. Moreover, we show the pre and post conditions of the use case. Finally, we go into the main success scenario that covers what the primary actor has to do to enable the use case as well as exceptions that need to be considered.

4.2.1 Initiate Negotiation

Use Case	**Initiate Negotiation**
Primary Actor:	Negotiation Participant
Stakeholders and Interests:	• Consumer: Find good providers • Provider: Offer a good service
Description	To initiate a negotiation the Negotiation initiator has to deploy a smart contract. That contract provides security and integrity for the offers stored inside the contract. It also allows the negotiation partners to realize a negotiation strategy.
Example	A consumer needs access to compute resources and discovers a provider that has plenty of resources. The consumer, however, does not trust the provider and wants to negotiate SLAs to manage risk. In order to be able to negotiate with the consumer a contract needs to be deployed with a reference to the provider (his address).

Preconditions:	The precondition to start the negotiation is for the consumer to have found a suitable provider or for the provider to have a suitable offer for a consumer. For this the discovery process already needs to have taken place. Further, the provider and the consumer need to have Ethereum addresses and enough funds to perform transactions.
Postconditions:	A new contract will be created that can be used by the negotiation initiator and the negotiation responder to save and negotiate offers.

Main Success Scenario:

1. The initiator communicates with the responder to recieve the responders address
2. The initiator choses a monitor agent
3. The initiator creates a new contract for the consumer and provider

Exceptions

1. The contract does not get added to the chain
2. Out of gas exception

4.2.2 Store Offers

Use Case	**Securely Store Offers**
Primary Actor:	Negotiation Participant
Stakeholders and Interests:	• Consumer: Integrity • Provider: Customer Satisfaction

Description	To provide the benefits that a Blockchain brings to the negotiation process it is necessary to store the offers or at least a reference to the offers in the Blockchain. In our case, we store a reference to an offer in the Ethereum Blockchain. The reference that is created comes from IPFS, which returns a hash of the data. This provides integrity and provenance to the negotiation process. It further allows us to use smart contracts to implement a negotiation strategy.
Example	A consumer wants to send an offer to a provider. Instead of sending the offer directly to the consumer stores the offer in the Ethereum Blockchain and sends the provider the means to access that offer.
Preconditions:	The Negotiation Participants needs to have Ethereum accounts that are funded. A negotiation contract has to exist.
Postconditions:	A reference to an offer is stored in the Ethereum Blockchain. That reference is a IPFS hash.

Main Success Scenario:

1. Create an offer
2. Create a transaction with that offer
3. Get the receipt once the transaction is mined

Exceptions

1. Out of gas exception
2. Connection to an ipfs or ethereum node failed

4.2.3 Negotiate

Use Case	**Negotiate**
Primary Actor:	Negotiation Participant
Stakeholders and Interests:	• Consumer: aquire a service with the desiered functional and non-functional attributes. • Provider: maximize profit and utilized capacity.
Description	Each negotiation partner has a utility function. They seek to optimize their utility function and in order to do that multiple rounds of negotiation are necessary. This use case lets Negotiation Partners create counteroffers.
Example	A provider receives an offer from a consumer. The provider agrees with the functional attributes but does not agree with some of the non-functional attributes. Instead of rejecting the agreement the providers sends a counteroffer to the consumer.
Preconditions:	An offer needs to have been be created for which a counteroffer can be made. Further, a negotiation contract also needs to have been created. In order for a participant to negotiate a connection to IPFS and Ethereum needs to be established.
Postconditions:	The state of an offer is changed to resemble the counteroffer made.

Main Success Scenario:

1. A Negotiation Partner analyses the offers from the Negotiation Contract
2. A counteroffer is created and sent to the Blockchain
3. The counteroffer changes the state of the offer

Exceptions

1. No contract exists
2. Invalid state change
3. Out of gas

4.2.4 Referee

Use Case	**Monitor Service**
Primary Actor:	Referee
Stakeholders and Interests:	• Consumer: the service is delivered as intended and promised • Provider: customer satisfaction
Description	Many cloud providers promise to deliver a service to their consumer. Unfortunately what they promise is not always what they deliver. The negotiation initiator can choose a referee that can monitor and warn the provider with a yellow and a red flag. This should indicate to the provider if the SLA terms are fulfilled.

Example	The provider is delivering a service to a consumer and in the SLA a response time for a support ticket to a critical bug should be no more than 2 hours. However, the provider needed five hours to respond. The referee will compare the actual with the expected values, and depending on the severity report with a red or a yellow flag.
Preconditions:	A Neogtiation Contract needs to exist with an offer that is in the state $x \in \{Binding, Deposited\}$. An offer in the state x can also be referred to ans an agreement.
Postconditions:	The state of the agreement has changed with a different value for the flag variable.

Main Success Scenario:

1. The referee constantly compares the provided service with the SLA specified service
2. If the provided and the expected differ, the referee can flag the agreement
3. The referee flags yellow for moderate and red for severe

Exceptions

1. Out of gas exception
2. Invalid flag transition

5 Technology Stack

In this chapter, we address the technology used and the implementation of our system. Our framework should be regarded as a starting point for verifiable negotiation of Service Level Agreements (SLAs) between trusted and untrusted peers without the need for human intervention. The primary focus will be placed on incorporating a Blockchain into the negotiation and agreement process. We build our solution on the WS-Agreement Negotiation framework.

We take a different approach from what Pittl et al. (2018); Fill u. Härer (2018) did in their papers. Instead of developing a new domain-specific Blockchain, we use an existing Blockchain solution called Ethereum. It uses a Turing complete language called solidity that compiles down to bytecode and can be run on the Ethereum Virtual Maschine (EVM) Wood (2017). We do not concern ourselves with the design and development of a Blockchain but instead concentrate on the creation of Smart Contracts. The concepts and code described here can be used for other Blockchains that support Turing complete programming languages. We chose the Ethereum Blockchain as it is already battle tested[16] and has an active community. What we give up by using an existing Blockchain solution is the possibility to optimize for our domain-specific use case.

The contributions of this chapter include:

- Integrity of the complete negotiation process
- Contract escrow with a dispute function
- Decentralized neogitation without the need for third parties

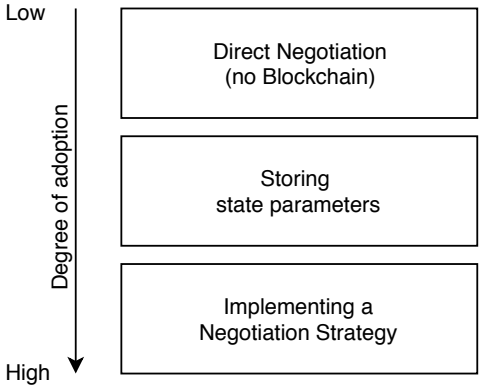
5.1 Layers of Adoption

In this section, we describe the extent to which our framework can be adopted. There are three layers of adoption as seen in Figure 5.1. The first layer represents a classical negotiation as in the WS-Agreement Negotiation standard. The second layer represents a negotiation where

[16] https://www.coindesk.com/understanding-dao-hack-journalists/ [30.08.18]

state variables are stored in the Blockchain. In the third layer not only are state variables stored but a negotiation strategy can be implemented within the Blockchain.

Figure 5.1: This diagram shows an overview of three layers from a low degree of adoption to a high degree of adoption



The last layer utilizes Smart Contracts that allow for predetermined code execution on the Blockchain. The distinction between the three layers is made to demonstrate the tradeoffs of integrating a negotiation strategy within a Blockchain.

Using a Blockchain is currently still very expensive. When choosing to adopt a Blockchain solution for a use case, this needs to be considered. Further, storing parameters to account for what happens leaves a smaller attack surface than integrating the negotiation strategy where decisions are made based upon the stored parameters. A layer three solution depends significantly on the so-called oracle providing the information to the Smart Contract. It is essential to consider who the oracles are when implementing a layer three solution. We demonstrate both the layer two and the layer three solution in our implementation.

5.2 Definitions

In this section, we describe the definitions of the terms that we use to extend the WS-Agreement Negotiation standard. All of the other terms and port types can be found in the papers of Waeldrich et al. (2011); Andrieux et al. (2007) please take a look if you are unsure what the meaning of an item is.

5.2.1 Verifiable Negotiation

The key to the negotiation approach that we describe in our framework is that it is verifiable. All of the publicly available information can be verified by both negotiation partners and any third party that is interested. The Negotiation Contract saves the state transitions of the negotiation process that can later be looked at by interested parties to ensure that the negotiation is fair. Moreover, monitoring can be conducted by a third party to ensure that the consumers get the value they were promised in the agreements.

5.2.2 Negotiation Contract

The Negotiation Contract is the contract that gets created by the Negotiation Initiator to negotiate with the Negotiation Responder. It contains a set of variables, functions, and modifiers that allow for negotiation participants to interact with the Blockchain. The variables represent the state, the functions represent the mechanisms to change the state, and the modifiers define the constraints of who is allowed to modify the state. Although state changes in the contract can , the previous state will still be documented and will not be lost.

5.2.3 Resource Address

The Resource Address (RA) is the Ethereum address of a Negotiator. The Ethereum address is a more digestible version of the public key. This address is necessary such that the negotiation participants can be identified within the created contract. The public key is the part of the key pair which should be publicly available for others, and the private key part should only be known to its owner.

5.3 Negotiation Contract

The structure of the contract depends mainly on security and cost considerations. The contract is compiled to an Application Binary Interface (ABI) and is then stored by each node in the Blockchain. Creating a transaction costs an additional creation fee of 32000 gas Wood (2017). We concluded that only relevant information that was needed to check conditions should be stored in the contract.

The Smart Contract gets deployed by the Negotiation Initiator and provides the following functionality:

- Store the hash of offers and agreements
- Retrieve the hash of offers and agreements
- Get an event when a new offer or agreement is stored
- Store additional meta data if necessary
- Provide access controls, such that only privileged users can store values.

As mentioned a big challenge with smart contracts is cost. Storing information in a Blockchain is extremely expensive, far more expensive than storing information in a traditional database. For exactly that reason we propose to only store the hash of the WS-Agreement XML file in the smart contract instead of the complete source file.

5.3.1 Parameters

In this section, we define the contract specific terms that are relevant for our implementation. We tried to keep the information stored by in the contract as minimal as possible to reduce the amount of gas that a transaction costs. Only the most relevant information should be included in the contract from the actual negotiation offer. The information we chose such as the Duration and the Deposit is necessary for limiting the set of people who can call specific function of the contract. Below we define the variables we chose to highlight and store in the smart contract.

Duration

The Duration is identical to the Context term in the WS-Agreement Expiration Time. However, it has two different meanings depending on the state of the offer. Once the offer is Binding it becomes an agreement and the current block time is added to the duration. This allows for a more dynamic negotiation as the duration first represents the time in seconds for how long the resource is needed and once a binding agreement is formed the duration represents an Expiration Time. The duration can be used by the withdraw and dispute functions of the contract to prevent a withdrawal before the service is provided and to prevent a dispute after the service has already been provided.

Deposit

The deposit is the cost of the resulting agreement. It is the price that the consumer has to pay for the resources that are made available. The variable is called Deposit as the money does not go directly to the provider but is instead held by the contract. The contract can hold onto the money until a specific condition is fulfilled.

5.3.2 State transitions

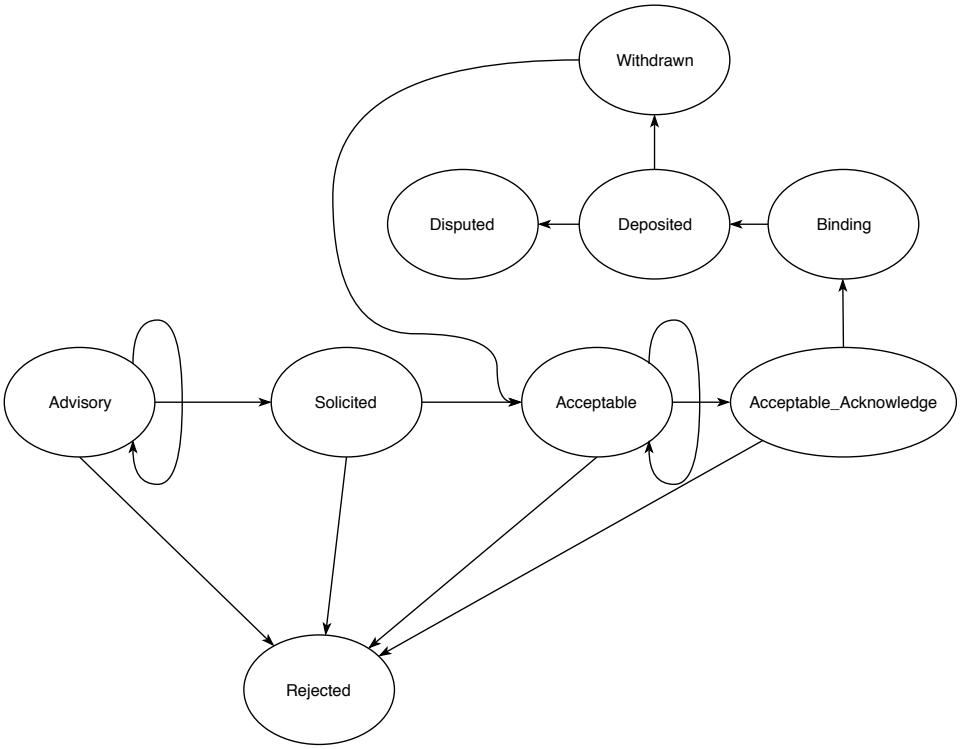
The states that an offer can be in are described by Waeldrich et al. (2011). The paper describes four states that an offer can be in before it becomes an agreement. These four states were not sufficient for autonomous negotiation as the Acceptable state did not necessarily mean that an agreement is going to be formed. For that reason Werner (2017) extended the states from four to six by introducing two new states that made it compulsory to create an agreement for a given offer. Further, to take advantage of smart contracts, we introduced three new states. The first being Deposited then Disputed and the third Withdrawn.

Table 5.1: An overview of the different states and their short cuts

State	**Short**
`Advisory`	Adv
`Solicited`	Sol
`Acceptable`	Acc
`Rejected`	Rej
`Accept_Acknowledge`	AcA
`Binding`	Bin
`Deposited`	Dep
`Disputed`	Dis
`Withdrawn`	Wit

We can define the possible state transitions formally as $S = (Q, \Sigma, \delta, q_0, F)$ where S is the state machine, Q are the possible states, Σ are the possible inputs, q_0 is the starting state and F are the possible final states. With this definition we get a set of possible states to be $Q = \{Adv, Sol, Acc, Rej, AcA, Bin, Dep, Dis, Wit\}$. The inputs are function calls to the contract with the next state as an argument $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. The start state is given by $q_0 = Adv$, and the possible final states by $F = \{Rej, Dis\}$.

Figure 5.2: An overview of the states that a contract can take as an extension to the WS-Agreement and the work of Werner (2017)

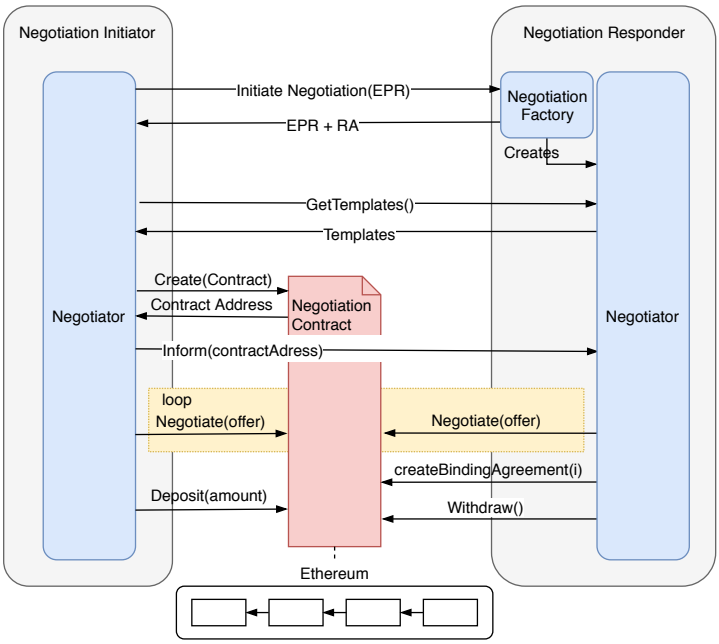


From Figure 5.2 we can see how the different state transitions can occur. The addition of $\{Deposited, Disputed, Withdrawn\}$ allows for the contract to determine if a function call is permissible or not. Certain function calls can only be executed if the offer or agreement is in the right state. For example, it is not possible to call the deposit function as long as the offer is in the Advisory state.

5.4 Bilateral Negotiation Model

The WS-Agreement Negotiation standard by Waeldrich et al. (2011) describes different negotiation designs, a simple client-server negotiation, bilateral negotiation, and re-negotiation. We focused on bilateral negotiations of agreements. It allows for asymmetric offer and agreement creation. Both the provider and consumer should be able to view the current state of the Negotiation Contract and be able to create a new Offer or be able to create a counteroffer.

Figure 5.3: Bilateral negotiation: an overview of how the smart contract is integrated into the negotiation process. The Endpoint Reference (EPR) and Responder Address (RA) are necessary for the Negotiation initiator to create the Negotiation Contract.



It is possible to deposit the price of the service into the smart contract. It does not go directly to the provider but instead the contract with act as a trustee. Only once the time for which the contract is valid has run out can the provider withdraw the money. If however, the provider does not comply with the negotiated terms it is possible for the consumer to change the state of the agreement to Disputed. In that case, an additional time penalty is added, which does not allow the provider to withdraw the money until the penalty has elapsed. The extra time prevents the provider from running off with the money and enables the consumer to take action against the provider and to settle the dispute. Other forms of dispute resolution would be to refund the consumer with the full amount that he deposited.

5.5 Implementation

In this section, we go into more detail about the technologies that we chose. Let us start looking at the Web Services Agreement Specification. We chose this specification because it offers a standard for negotiation that can be extended. The specific use case that our proposal builds upon is the bilateral negotiation from the WS-Agreement Negotiation framework. We propose

to enhance bilateral negotiation by adding a trustless storage capability, to the negotiation process. The storage capability is comprised of two technologies. The first is the Inter Planetary File System Benet (2014), which uses a Merkle DAG object storage, to keep the integrity of the system and quickly access files. The second part is a Blockchain, namely Ethereum, where we store the hash of the offer(s) and agreement(s). The Blockchainâs key addition to IPFS is able to track provenance and ownership. Previously, each participant in the negotiation was themselves responsible for safely storing agreements. There was no way of definitively proving what agreement was made, without a trusted third party. Our implementation is open source and can easily be reproduced[17].

5.5.1 Dependencies

We will give a brief overview of the most important dependencies used by our implementation. For a more detailed overview visit our github page in the footnote below.

Parity

Parity[18] is an Ethereum client written in the Rust programming language. Rust is a systems language with a focus on safety. Parity comes as a CLI tool and can expose a JSON-RPC HTTP API via the port 8545. This API can be used to interact with the Ethereum blockchain. Partiy relies on openssl to create new accounts and to sign transactions.

IPFS

The quote below sums IPFS[19] up very well.

> IPFS is a peer-to-peer hypermedia protocol to make the web faster, safer and more open.

It is essentially a decentralized file system that benefits from the concepts brought forth by git and BitTorrent. It uses a distributed hash table to index the files and a pruning algorithm that removes duplicate hashes. IPFS can be accessed over HTTP and the content is spread across many nodes.

[17] https://github.com/qu0b/conviction
[18] https://parity.io
[19] https://ipfs.io [23.09.18]

Nodejs

Initially, Javascript would only be run in the browser until the Nodejs project was first released. It is built on Chromes V8 Javascript engine and compiles at runtime. The language only uses a single thread for its processes but takes advantage of asynchronous programming paradigms. Nodejs comes with a widely popular package manager called npm. Nodejs allows for the maximum code reusability, which is the reason we used Nodejs. This way our framework could potentially also be used as a distributed app accessible via the browser.

Web3

This is a client library which communicates with the Ethereum node, in our case parity, over RPC calls. The man components of the library that we use are web3.eth.Contract, web3.eth.accounts and web3.eth.personal. The web3 package was added to our implementation as a dependency.

Solc

This package is the solidity compiler that allows us to compile smart contract code and use it with the Web3 client. The compiler generates the contract Abstract Binary Interface (ABI) and the bytecode which is necessary to deploy a contract to the Ethereum blockchain.

5.5.2 Decentralized Storage

We decided to use IPFS to reduce the cost of storing offers and agreements. Further, we are reluctant to store the information centrally as the point of using a Blockchain is to be decentralized. IPFS allows for decentralized storage and indexes information with a hash value. We store the XML documents using IPFS and then store the resulting hash value in the contract. The ipfs_reference variable was thus added to the smart contract, which allows the negotiation participants to store a reference to the XML version of the SLA negotiation offer.

Figure 5.4: The steps necessary to store an offer received from a negotiation participant

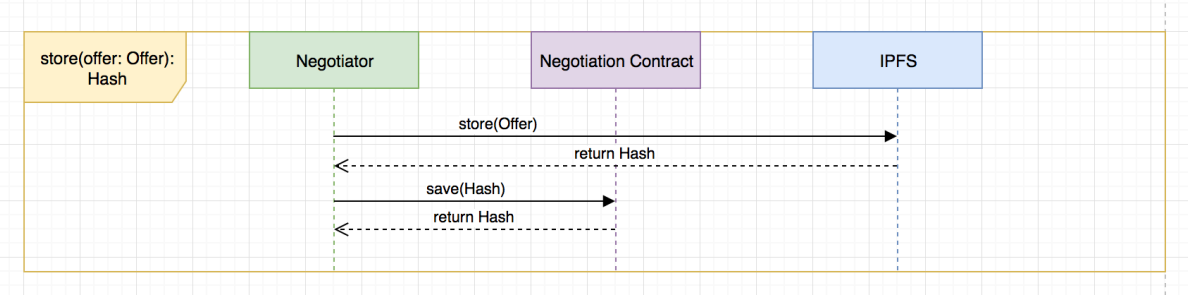
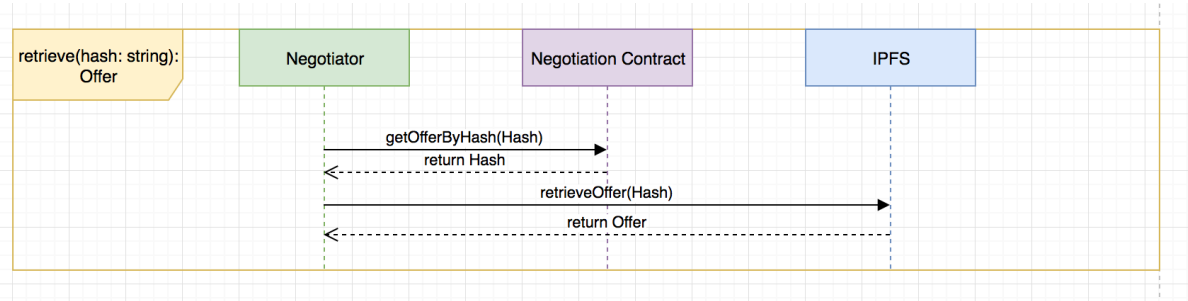


Figure 5.5: The steps necessary to retrieve the source of an offer from the IPFS Merkle Hash



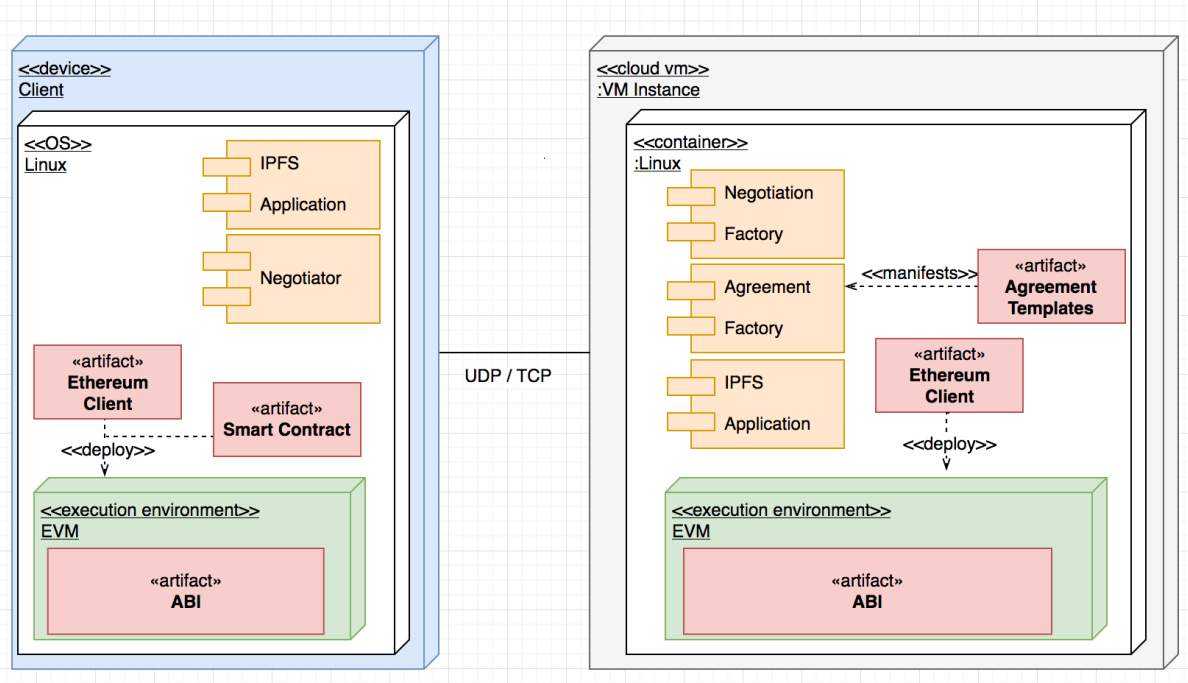
To visualize the process, we created two diagrams 5.4 and 5.5 that show how the proposed storage and retrieval capabilities look like. The lifelines in the diagram show the negotiator, the negotiation contract and the IPFS file system. When the function store is called the source of the offer is stored in the IPFS. The IPFS adds the source of the offer to a Merkle DAG and returns a hash value. The hash value is then stored in the smart contract so that a negotiation partner can view and retrieve the file. The retrieval process 5.5 works similar but in reverse. First, the negotiator looks up the hash from the smart contract, then searches in the IPFS for the source of the offer.

5.5.3 Deployment

For a better understanding of what hardware, software, and artifacts are necessary to realize the negotiation we created a deployment diagram 5.6. On the left side, there is the consumer, this can be any IoT device or similar, and on the left side, there is a provider. The client has two components, which are the IPFS application and the Negotiator. The provider has similar components, the difference being that there is a negotiation factory that can create negotiators on demand for the provider. Additionally, the provider has an Agreement Factory which offers

the agreement port types from the WS-Agreement Negotiation. The smart contract is deployed by the Ethereum Client, which results in an Application Binary Interface file that gets executed on the Ethereum Virtual Machine (EVM).

Figure 5.6: The deployment diagram shows an overview of the exact hardware and software. The left depicts a consumer that is negotiating with a Negotiation Instance of a provider.



5.5.4 Application Interface

Here we describe the server API that was created in order to interact with the Smart Contract. The server runs on NodeJS and talks to both the IPFS node and Ethereum node running on the system. The POST operations are described as they define the important state changes that the server can make. The GET operations should be self-explanatory from the URL structure.

Table 5.2: This table shows the structure used for the request response descriptions below.

HTTP METHOD	**ENDPOINT[?parameter(s)]**
HEADER(s)	Array<{key: value}>
Body	[Object]
RESPONSE	[Object]

Create Contract

This method supplies the necessary parameters for the server to create a new contract. It is necessary for the initiator to have the address unlocked or authenticated so that the server can create the contract in the initiators name.

POST	**/create/contract**
HEADER(s)	{"content-type": "application/json"}
Body	"initiator":<address>, "responder": <address>, "referee": <address>, "isConsumer": <bool>
RESPONSE	"result": <contract>

Create Offer

This endpoint allows a Negotiation Partner to create an offer. An existing contract is necessary for an offer to be created. The address is necessary for the server to sign the transaction. If the account is not unlocked a *pass* variable can be included in the Request body.

POST	**/contract/:contractId/offer**
HEADER(s)	{"content-type": "application/json"}
Body	"address":<address>, "offer":<xml>, "deposit": <amount>, "duration": <unixTime>
RESPONSE	"result": <transaction>

Make a Counter Offer

For existing offers, it is possible to create a counteroffer. The counteroffer endpoint needs the counter offer XML file, and it needs the new state. Further, the request can optionally have the deposit and the duration specified to override the existing values of the offer. The transaction keyword refers to the transaction receipt that gets returned from the Ethereum Blockchain.

POST	**/contract/:contractId/offer/:offerId**
HEADER(s)	{"content-type": "application/json"}
Body	"address":<address>, "counterOffer":<xml>, "state": <number> "deposit"?: <amount>, "duration"?: <unixTime>
RESPONSE	"result": <transaction>

Create Agreement

Once a negotiation has reached the state Accept_Acknowledge it is possible for a negotiation participant to create an agreement.

POST	**/contract/:contractId/offer/:offerId/createAgreement**
HEADER(s)	{"content-type": "application/json"}
Body	"address":<address>, "agreement":<xml>
RESPONSE	"result": <transaction>

Deposit Funds

As soon as the offer becomes an agreement it is possible for the consumer to deposit the funds. The agreement goes from state Binding to state Deposited and the provider can be sure that the consumer has the necessary funds.

POST	**/contract/:contractId/offer/:offerId/deposit**
HEADER(s)	{"content-type": "application/json"}
Body	"address":<address>, "value":<amount>,
RESPONSE	"result": <transaction>

Withdraw Funds

Once the agreement duration has run out it is possible for the provider to withdraw his funds by calling this endpoint.

POST	**/contract/:contractId/offer/:offerId/withdraw**
HEADER(s)	{"content-type": "application/json"}
Body	"address":<address>,
RESPONSE	"result": <transaction>

Flag Agreement

For the referee, it is possible to call the flag endpoint with the specified number. The number specifies the index of the flag enumeration. $num \in \{0 : none, 1 : yellow, 2 : red\}$

HEADER(s)	{"content-type": "application/json"}
Body	"address":<address>,
POST	**/contract/:contractId/offer/:offerId/flag/:num**
RESPONSE	"result": <transaction>

Raise a Dispute

It is possible for the participant to dispute the agreement. In that case, an arbitrary penalty gets executed. In the case of our implementation, the duration gets increased, and the provider needs to wait for a longer duration to withdraw his funds.

POST	**/contract/:contractId/offer/:offerId/dispute**
HEADER(s)	{"content-type": "application/json"}
Body	"address":<address>,
RESPONSE	"result": <transaction>

5.5.5 Contract Code

The smart contract code has been added to get an overview of the functions that can be provided. In the dispute function, two possible repercussions are defined. It is possible to adapt the contract as it is a reference implementation.

Simple Negotiation

This contract can be used for storing the IPFS hash in the Blockchain. It does not realize any other logic apart from only letting the Negotiation Partners call the functions offer and counteroffer.

```
1   contract SimleNegotiation {
2   string[] public ipfs_references;
3   address initiator;
4   address responder;
5
6  constructor(
7       address _responder
8   ) public {
9     initiator = msg.sender;
10    responder = _responder;
11  }
12
13  function offer(string _ipfs_reference) onlyParticipant returns (uint) {
14    uint id = ipfs_references.push(_ipfs_reference);
15    return id;
16  }
17
18  function counterOffer(uint index, string _ipfs_reference) onlyParticipant {
19    ipfs_references[index] = _ipfs_reference;
20  }
21
22    modifier onlyParticipant() {
23    require(msg.sender == initiator || msg.sender == responder);
24    _;
25  }
26 }
```

Negotiation Strategy

This Smart Contract implements logic that allows for the negotiation and the storage of offers and agreements. It implements a referee and a dispute functionality to improve the provided service.

```
1  pragma solidity ^0.4.23;
2
3  contract Negotiation {
4    enum States { Advisory, Solicited, Acceptable, Rejected,
       Acceptable_Acknowledge, Binding, Deposited, Disputed, Withdrawn }
5    enum Flags { none, yellow, red }
6
7    address initiator;
8    address responder;
9    Offer[] public offers;
10   mapping(uint => address) payee;
11   event depsoitMade(address consumer, uint amount);
12   event agreementMade(uint index, uint end, uint deposit);
13   event stateChange(uint index, uint8 next);
14   event newOffer(uint index, string ipfs_reference, uint deposit, uint
       duration);
15   address referee;
16   bool isConsumer;
17
18    struct Offer {
19     uint id;
20     address creator;
21     string ipfs_reference;
22     uint deposit;
23     uint duration;
24     States state;
25     Flags flag;
26   }
27
28   constructor(
29       address _responder,
30       address _referee,
31       bool _isConsumer
32   ) public {
33     initiator = msg.sender;
34     responder = _responder;
35     referee = _referee;
36     isConsumer = _isConsumer;
37   }
```

```
38
39      function offer(
40        string _ipfs_reference,
41        uint _deposit,
42        uint _duration
43      ) public onlyParticipant returns (uint) {
44        uint length = offers.length;
45        offers.push(Offer({
46          id: length,
47          creator: msg.sender,
48          ipfs_reference: _ipfs_reference,
49          deposit: _deposit,
50          duration: _duration,
51          state: States.Advisory,
52          flag: Flags.none
53        }));
54        emit newOffer(length, _ipfs_reference, _deposit, _duration);
55        return length;
56      }
57
58      function counterOffer(
59        uint _responseTo,
60        string _ipfs_reference,
61        uint8 _state
62        ) public onlyParticipant validStateTransitions(offers[_responseTo].state,
              States(_state)) {
63        require(offers[_responseTo].creator != msg.sender);
64        offers[_responseTo].creator = msg.sender;
65        offers[_responseTo].state = States(_state);
66        offers[_responseTo].ipfs_reference = _ipfs_reference;
67        emit stateChange(offers[_responseTo].id, _state);
68      }
69
70      function counterOffer(
71        uint _responseTo,
72        string _ipfs_reference,
73        uint _deposit,
74        uint _duration,
75        uint8 _state
76        ) public onlyParticipant validStateTransitions(offers[_responseTo].state,
              States(_state)){
77        require(msg.sender != offers[_responseTo].creator);
78        offers[_responseTo].creator = msg.sender;
79        offers[_responseTo].duration = _duration;
80        offers[_responseTo].deposit = _deposit;
```

```
81        offers[_responseTo].state = States(_state);
82        offers[_responseTo].ipfs_reference = _ipfs_reference;
83        emit stateChange(offers[_responseTo].id, _state);
84      }
85
86      function createAgreement(
87        uint _responseTo,
88        string _ipfs_reference
89        ) public onlyParticipant {
90          require(offers[_responseTo].state == States.Acceptable_Acknowledge);
91          require(msg.sender != offers[_responseTo].creator);
92          offers[_responseTo].creator = msg.sender;
93          offers[_responseTo].state = States.Binding;
94          offers[_responseTo].ipfs_reference = _ipfs_reference;
95          offers[_responseTo].duration = now + offers[_responseTo].duration;
96          emit agreementMade(offers[_responseTo].id, offers[_responseTo].duration
                , offers[_responseTo].deposit);
97      }
98
99      function deposit(uint index) public payable {
100       require(isConsumer && msg.sender == initiator || !isConsumer && msg.
              sender == responder);
101       require(offers[index].state == States.Binding);
102       require(msg.value == offers[index].deposit, 'Wrong deposit amount.');
103       payee[index] = msg.sender;
104       offers[index].state = States.Deposited;
105       emit depsoitMade(msg.sender, msg.value);
106     }
107
108     function withdraw(uint index) public onlyParticipant returns (bool) {
109       require(msg.sender != payee[index]);
110       require(offers[index].state == States.Deposited, "First deposit money");
111       require(now > offers[index].duration, "Agreement not expired");
112
113       if(offers[index].deposit > 0) {
114         if (msg.sender.send(offers[index].deposit)) {
115             offers[index].state = States.Withdrawn;
116             offers[index].deposit = 0;
117             return true;
118           } else {
119             return false;
120           }
121       } else {
122         return false;
123       }
```

```
124     }
125
126     function dispute(uint index) public {
127       require(msg.sender == payee[index]);
128       require(offers[index].state == States.Deposited);
129       require(now <= offers[index].duration);
130       offers[index].state = States.Disputed;
131       // offers[index].duration += 604800; // adds a week
132       payee[index].transfer(offers[index].deposit); // transfers the funds back
              to the payee
133     }
134
135     function setFlag(uint index, uint _flag) public {
136       require(msg.sender == referee);
137       Flags flag = Flags(_flag);
138       require(Flags.red == flag || Flags.yellow == flag);
139       offers[index].flag = flag;
140     }
141
142     modifier onlyParticipant() {
143       require(msg.sender == initiator || msg.sender == responder);
144       _;
145     }
146
147     modifier validStateTransitions(States previous, States next) {
148       require(previous != next || previous == States.Advisory && next == States
              .Advisory || previous == States.Acceptable && next == States.
              Acceptable);
149       bool advToSol = previous == States.Advisory && next == States.Solicited;
150       bool solToAcc = previous == States.Solicited && next == States.Acceptable
              ;
151       bool accToAck = previous == States.Acceptable && next == States.
              Acceptable_Acknowledge;
152       bool solToRej = previous == States.Solicited && next == States.Rejected;
153       bool accToRej = previous == States.Acceptable && next == States.Rejected;
154       bool ackToRej = previous == States.Acceptable_Acknowledge && next ==
              States.Rejected;
155       bool witToAcc = previous == States.Withdrawn && next == States.Acceptable
              ;
156       require(advToSol || solToAcc || accToAck || solToRej || accToRej ||
              ackToRej || witToAcc, 'invalid state transition');
157       _;
158     }
159   }
```

6 Use Case

In this chapter, we describe a scenario and go through two concrete examples of that scenario using our framework. The scenario describes a high-level overview and the practical examples go into more detail. The first example depicts a negotiation with multiple offers and counteroffers while only storing a reference in the Blockchain. In the second example we put more emphasis on the state transitions as that example uses a Smart Contract with more logic.

There are many instances in which a device, especially small devices such as IoT devices, would need access to computing resources. For example, small battery driven devices, which need to execute compute heavy operations, could outsource their operations to the cloud and save battery power in the process. However, the devices could be spread across different availability zones, which makes it hard for a single broker to provide adequate quality of service terms. The devices need to be able to negotiate on demand with providers that are available.

A small battery powered IoT camera could submit a video encoding task to a cloud provider. Without the need to trust the provider the camera could depending on a predefined utility function negotiate with a provider. Once the negotiation is complete and a binding agreement is formed neither party can falsify the result. Monitoring entities can compare the provided service to what was promised by the agreement. If the monitoring entity discovers a discrepancy, it can record said discrepancy, indirectly warning the provider. In the case of untrusted peers, a fallback mechanism can be used. This fallback mechanism would allow the consumer to dispute the agreement. Since the payment is not made directly to the provider but is diverted through the contract, such a dispute would mean that the funds could be held hostage or returned to the consumer.

If instead of one camera we were talking about thousands it would be impossible to individually negotiate a SLA for each. A broker would be needed to negotiate on behalf of the cameras. The provenance and dispute functionality of our framework lets IoT devices negotiate directly with a cloud provider without using a broker. Using a shared automatable ledger allows negotiation to occur autonomously such that devices can provision resources without the need for human intervention.

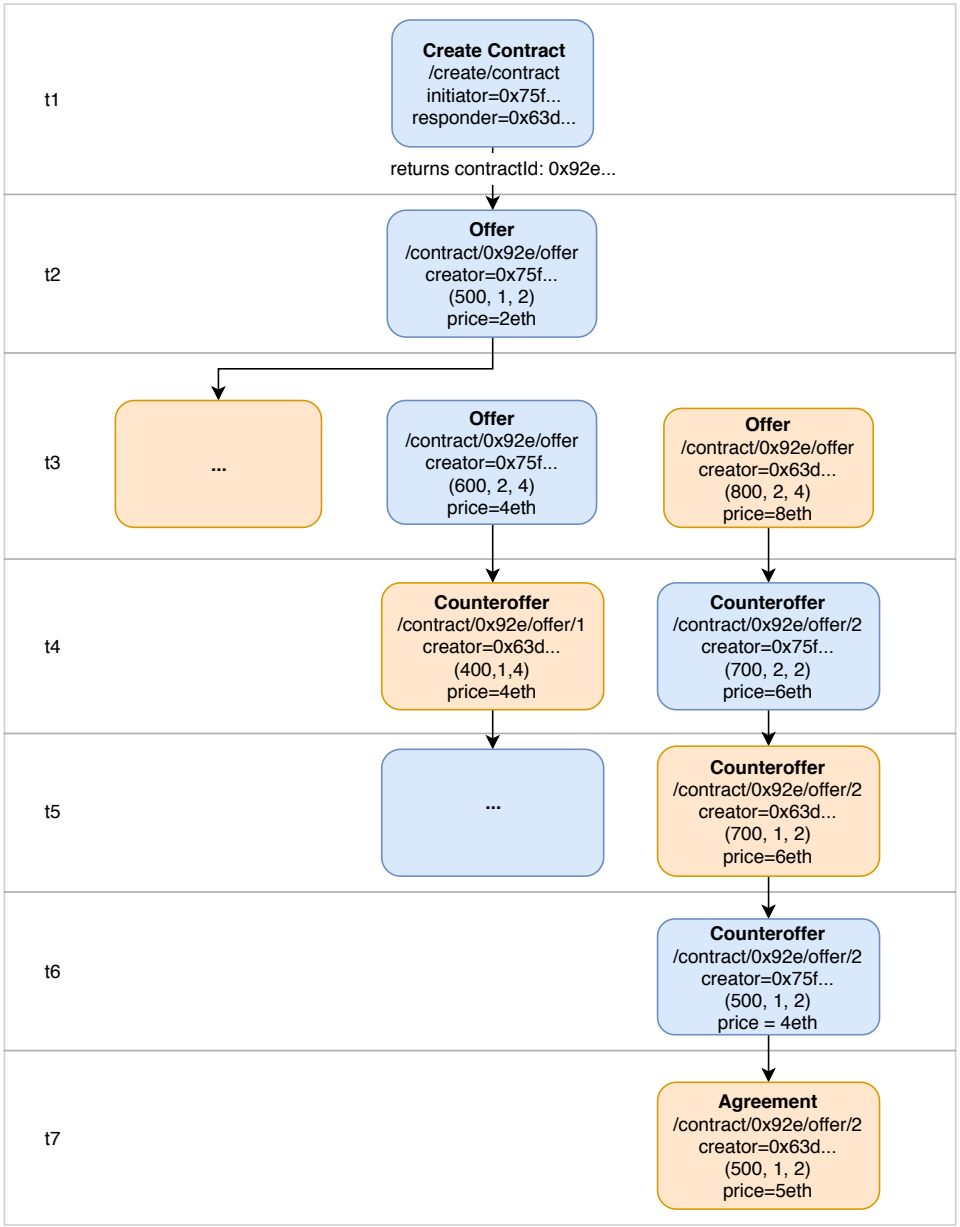
Our camera needs to negotiate a SLA with a provider such that the camera has enough storage to host all of its videos, images, and provide computing capacity to convert its videos into different formats. For that, the camera tries to acquire a virtual machine with $500GB$ of storage, $1vCPU$ core and $2GB$ of RAM. In case the camera runs out of battery the server needs to run for a long enough period so that the camera has time to charge. The duration will be 3 weeks and the price, that should not be exceeded, is $5eth$, eth is short for Ether the currency of the Ethereum. The utility function of the camera would then be $500x + y + 2z \leq 5$ where x is the price of a GB of storage, y is the price of a $vCPU$ and z is the price of a GB of RAM, which can also be written in tuple form (500, 1, 2). Further, there are quality of service terms, the provider needs to have a higher than 99.9% service level and a maximum latency of 200ms. The price is the quantity that gets deposited for the provider to withdraw after the duration has expired. The utility function of the provider is $x = 0.0001$, $y = 1.8$, $z = 0.2$ on a per month basis (in eth).

6.1 Simple Negotiation

To understand how the negotiation process occurs with respect to time we created a diagram as seen in Figure 6.1. It gives an overview of how a negotiation process occurs with respect to time t. In each time step t the negotiation participant can execute a function that is directed at the Negotiation Contract. To set up the negotiation the Negotiation Initiator first talks to the Negotiation Factory port type to create a Negotiator for the Negotiation Responder. To create a contract the Negotiation Initiator needs to discover a Negotiation Responder and get their Resource Address (RA). Once the RA is acquired a contract is created and the negotiation can begin. Although before negotiation can begin the other participant needs to be informed as seen in 5.3. At time $t1$ the contract is created and the negotiation participants can create offers and counteroffers.

The Figure 6.1 is a negotiation where only the hash of the WS-Agreement file is stored in the Blockchain. Participants can create offers in parallel e.g. $t3$. The negotiation carries on, and an agreement can be formed. The diagram is created from the perspective of the Smart Contract. The negotiation occurs in a similar fashion as in the WS-Agreement negotiation apart from the fact that the offers are stored in a shared database.

Figure 6.1: A simple negotiation example of the offers and counteroffers over the time periods t. The resources in the offers are described in tuples (storage in GB, vCPU in cors, RAM in GB). The endpoints called are stated below the title of each node.



In the following example we add the concept of state transitions to the contract. The contract in this example cannot currently prevent invalid state transitions.

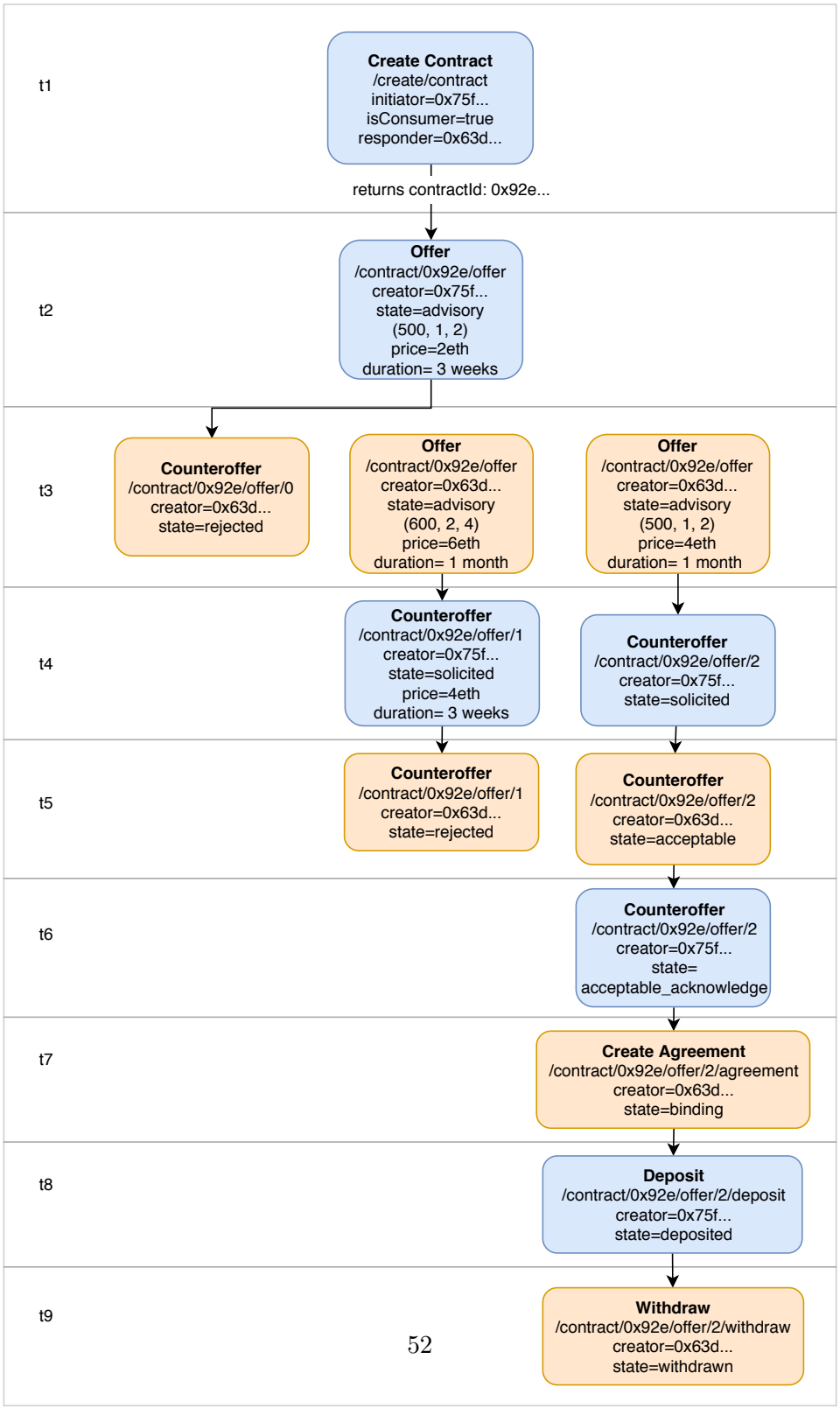
6.2 Negotiation Strategy

In this example, we use the same variables as in the previous one, but this time we will describe the implementation of a negotiation strategy via Smart Contracts. The initiator is in blue and has an Ethereum RA starting with *0x75f*. The consumer is in orange and has a RA starting with *0x63d*. Each arrow from one node to the next symbolizes an API call. The call is made to the endpoint in the former node. The contract used for this example stores the duration and the *deposit* (depicted as the price in the diagram) of the offers. These are two state variables among others, such as the initiator and the responder, that can be used in the contract to restrict API calls. In this example, we have a different endpoint for creating an agreement compared to the previous. This is due to the fact that we modify the *duration* state variable. Once an agreement is formed the duration becomes the expiration date. We add the current date to the duration e.g. $duration = now + duration$.

Each negotiation participant can always create new offers. Once an offer is in the accept_acknowledge state, the counterparty can form a binding agreement. This is different from the counteroffer as it semantically changes the offer to an agreement. After an agreement is made the consumer is expected to deposit the agreed upon price into the contract. When the consumer deposits the funds the contract changes state again to *deposited*. In this state the consumer can call a *dispute* functionality in case the referee, described in the next section, discovers a discrepancy. If everything goes well, and the consumer is provided with an adequate service the provider can withdraw the funds from the contract. This is only possible when the modified *duration* has expired. When the money is withdrawn the state of the agreement becomes *withdrawn* and the agreement can be used as a template for a new offer. On the other hand, if the agreement is *disputed* it can no longer be used again. The possibility to change state from *withdrawn* to *acceptable* was done to support renegotiation of successful agreements.

Since the negotiation is bilateral restrictions are in place to prevent illegal state changes. For example, once an offer is in the rejected state, its state can no longer be changed by a counteroffer. More trivial restrictions in place prevent a participant from creating two counter offers for the same negotiation branch. The possible state transitions can be seen by the Figure 5.2. If an invalid state change is made the transaction is reverted and the gas used is lost.

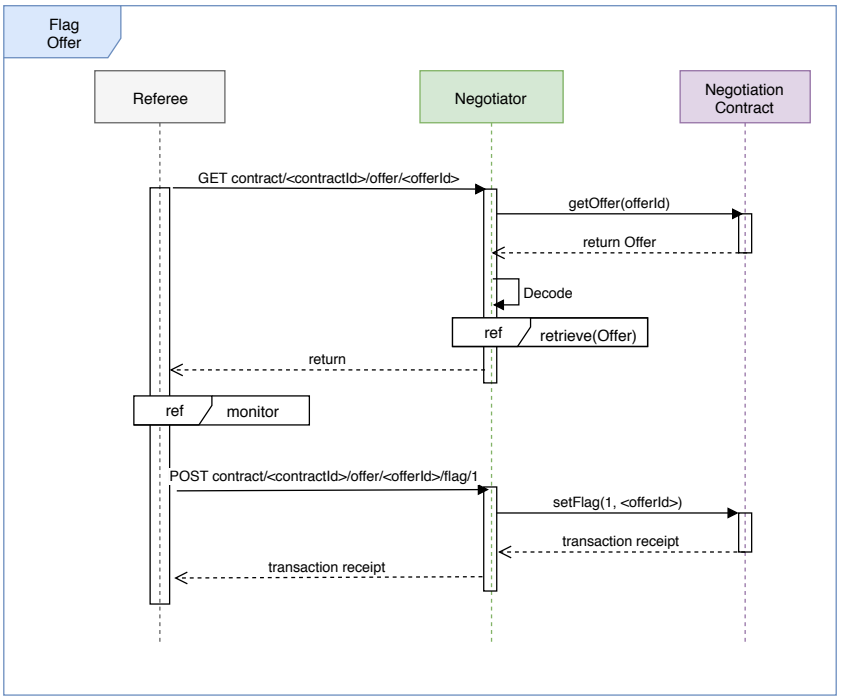
Figure 6.2: An example of the offers and counteroffers over the time periods t with an emphasis on state changes. The resources in the offers are described in tuples (storage in GB, vCPU in cors, RAM in GB). The endpoints called are stated below the title of each node.



6.3 Cloud Referee

Similar to the idea presented by Liu et al. (2011) of a cloud auditor a cloud referee is responsible for monitoring the provided service. The referee is an independent agent that ensures service objectives and quality of service terms are met. If the provider fails to provide an adequate service it is the referee's job to record the discrepancies and signal the provider.

Figure 6.3: In this diagram, we can see how a referee monitors and flags an offer. In the first step, the referee gets the offer and then monitors the offer by comparing the expected and delivered values.



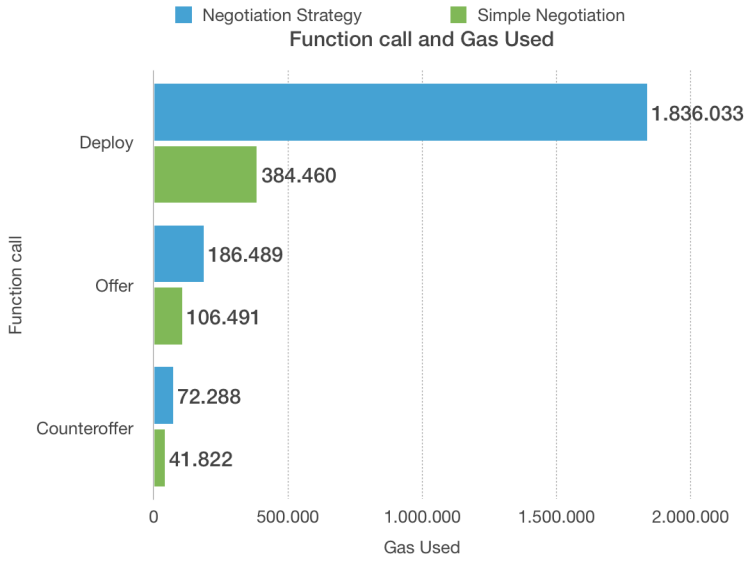
In the Figure 6.3 we can see how a referee can be used to flag an offer. The offer is first retrieved from the negotiation contract and is then decoded by the negotiator. The negotiator handles the retrieval of the hash encoded offer. The offer is stored in an IPFS node and the hash returned from the negotiation contract is used by the negotiator to retrieve the actual file. After the referee has the offer, he can perform a monitoring operation. The definition of the monitoring is not in the scope of this paper and is thus referenced. If the referee finds a fault, he can call the Negotiator and flag the offer.

7 Evaluation

The benefits of the blockchain come with some downsides. In some sense, a Blockchain is a costly database. If it is worth using a Blockchain depends on how high the stakes are. If the camera was monitoring a bank vault, then it might be worth spending Ether to negotiate an agreement, but otherwise, it might not be worth it. Below we will compare the costs of realizing a negotiation strategy using Smart Contracts to only storing the IPFS hash of the WS-Agreement file. We do this to show how the additional logic in the negotiation contract affects the cost.

What we did not account for in the utility functions, described in the previous chapter, are the gas costs. With the addition of Smart Contracts, each offer and counteroffer costs money. The result is a system in which the gas costs need to be considered by both participants. For example, we can consider the transaction costs as a constraint $gasused * gasprice \leq 1$. If the costs exceed the constraint, then the negotiation can be terminated.

Figure 7.1: This diagram shows an overview of the gas used by our implementation.



From Figure 7.1 we can see that the cost to deploy a contract is immense. That being said it

is important for the negotiation participants to use a single contract as often as possible and to renegotiate offers. Moreover, we can see that adding logic to a contract disproportionately increases the deployment cost immensely. We currently deploy a contract for each bilateral negotiation, but it would be worth a thought to create a generic contract with an initNegotiation function. If more logic is added to the contract the gas cost would be too high for a new contract to be deployed for each negotiation. Each block has a gas limit that caps the maximum gas that can be spent by one transaction. Very complex contracts could reach this limit. So instead of creating a new contract for each negotiation, it could be possible to create a library that can be used. The downside to this is that all of the negotiations are stored in one place, and it becomes more difficult to stay anonymous.

8 Future Work and Conclusion

In this chapter, we will discuss the results of our approach and how it could be extended. We also conclude the paper and present our opinion.

8.1 Future Work

The field of DLT is still very new, and there is a lot of activity. Many projects, other than Ethereum, argue that their Blockchain is better because they can e.g. process more transactions, but most of the time the improved feature that they advertise is a tradeoff with another important feature. In this section, we will discuss some of the recent work in the Blockchain ecosystem, and we will discuss methods how our approach could be extended.

There have been solutions to improve the number of transactions that can be processed with Ethereum and other Blockchains. These solutions do not propose to change the current architecture but instead to build another layer (a layer two) on top of the Blockchain. One of the solutions is called plasma by Poon u. Buterin (2017) where another, less secure, Blockchain with a different consensus algorithm (Proof of Stake) is built on top of the Ethereum Blockchain. This layer two Blockchain will have a fallback functionality and only save snapshots of the layer two (using concepts such as Merkle trees) into the underlying Ethereum Blockchain. A similar second layer technology was introduced for the Bitcoin Blockchain also by Poon u. Dryja (2016). These off chain payments use channels that need to be funded and can then settle any number of smaller transactions for a lower fee.

The next aspect that needs more research are oracles. These are the entities that write information onto the Blockchain. If, for example, we added a monitoring functionality the critical question to solve would be who is allowed to write to the contract and how do we handle malicious actors. On the other hand, it would be helpful to develop a proof of service algorithm that could be used by the provider to deliver evidence of the service to the smart contract. The current Blockchains do not have a fallback mechanism which makes the oracles powerful. The above mentioned second layer technologies help in that regard as they allow for a fallback mechanism to be put in place.

Another exciting research direction would be to look at the provisioning of containers and to look at how SLAs could be negotiated by the coordinator such as in Kubernetes[20] for individual and pools of containers. It would allow the autonomous provisioning to be combined with the autonomous negotiation for SLAs.

Possible further steps that can be taken:

- Optimize the gas cost of a negotiation
- Realize autonomous negotiation strategies that run on the Blockchain Baig et al. (2017)
- Negotiate and monitor SLAs for containers[21]
- Build a layer two Blockchain Poon u. Buterin (2017)
- Analyze the security of Smart Contracts

8.2 Conclusion

We looked at Service Level Agreements (SLAs), standards for negotiating SLAs and the Distributed Ledger Technology (DLT). More specifically we proposed a method for storing offers and agreements in a trustless manner where no third party would be needed. We also showed that there are methods of dispute resolution, and discovered that the bottleneck of SLA negotiation over the Blockchain is the cost of contract creation. The benefits of adding a Blockchain are evident although the transaction costs still need to be reduced in order to make the approach viable.

Blockchains make it possible to verify provenance and ownership, which makes SLA negotiation become more manageable. It allows realizing trustless autonomous negotiations, with the addition of referees that monitor the services. The Solidty language is turing complete, which allows for negotiation participants to extend and customize the framework. Lastly, we found that IPFS works well but still needs more adoption.

[20] https://kubernetes.io/
[21] https://kubernetes.io/

Bibliography

Andrieux et al. 2007

Andrieux, Alain ; Czajkowski, Karl ; Dan, Asit ; Keahey, Kate ; Ludwig, Heiko ; Nakata, Toshiyuki ; Pruyne, Jim ; Rofrano, John ; Tuecke, Steve ; Xu, Ming: Web services agreement specification (WS-Agreement). In: *Open grid forum* Bd. 128, 2007, S. 216 (cited on pages 2, 14, 19 and 30).

Androulaki et al. 2012

Androulaki, Elli ; Karame, Ghassan ; Roeschlin, Marc ; Scherer, Tobias ; Capkun, Srdjan: Evaluating User Privacy in Bitcoin. In: *Financial Cryptography*, 2012 (cited on page 9).

Androulaki et al. 2013

Androulaki, Elli ; Karame, Ghassan O. ; Roeschlin, Marc ; Scherer, Tobias ; Capkun, Srdjan: Evaluating user privacy in bitcoin. In: *International Conference on Financial Cryptography and Data Security* Springer, 2013, S. 34–51 (cited on page 10).

Armbrust et al. 2010

Armbrust, Michael ; Fox, Armando ; Griffith, Rean ; Joseph, Anthony D. ; Katz, Randy ; Konwinski, Andy ; Lee, Gunho ; Patterson, David ; Rabkin, Ariel ; Stoica, Ion et al.: A view of cloud computing. In: *Communications of the ACM* 53 (2010), Nr. 4, S. 50–58 (cited on page 1).

Back et al. 2007

Back, Adam et al.: *Hashcashâa denial of service counter-measure, 2002*. 2007 (cited on pages 8 and 9).

Baig et al. 2017

Baig, Ramsha ; Khan, Waqas A. ; Haq, Irfan U. ; Khan, Irfan M.: Agent-based SLA negotiation protocol for cloud computing. In: *Cloud Computing Research and Innovation (ICCCRI), 2017 International Conference on* IEEE, 2017, S. 33–37 (cited on pages 2 and 57).

Battré et al. 2010

Battré, Dominic ; Brazier, Frances M. ; Clark, Kassidy P. ; Oey, Michael ; Papaspyrou, Alexander ; Wäldrich, Oliver ; Wieder, Philipp ; Ziegler, Wolfgang: A proposal for WS-agreement negotiation. In: *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on* IEEE, 2010, S. 233–241 (cited on page 15).

Benet 2014

Benet, Juan: IPFS - Content Addressed, Versioned, P2P File System. In: *CoRR* abs/1407.3561 (2014) (cited on pages 5 and 36).

BitFury 2015

BitFury: Proof of Stake versus Proof of Work, 2015 (cited on page 17).

Blasi et al. 2013

Blasi, Lorenzo ; Brataas, Gunnar ; Boniface, Michael ; Butler, Joe ; DâAndria, Francesco ; Drescher, Michel ; Jimenez, Ricardo ; Krogmann, Klaus ; Kousiouris, George ; Koller, Bastian ; Landi, Giada ; Matera, Francesco ; Menychtas, Andreas ; Oberle, Karsten ; Phillips, Stephen ; Rea, Luca ; Romano, Paolo ; Symonds, Michael ; Ziegler, Wolfgang: *Cloud Computing Service Level Agreements – Exploitation of Research Results.* 2013 (cited on page 15).

Bramas 2018

Bramas, Quentin: *The Stability and the Security of the Tangle.* `https://hal.archives-ouvertes.fr/hal-01716111`. Version: April 2018. – working paper or preprint (cited on page 13).

Buyya et al. 2009

Buyya, Rajkumar ; Yeo, Chee S. ; Venugopal, Srikumar ; Broberg, James ; Brandic, Ivona: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. In: *Future Generation computer systems* 25 (2009), Nr. 6, S. 599–616 (cited on pages 1 and 2).

Dastjerdi u. Buyya 2012

Dastjerdi, Amir V. ; Buyya, Rajkumar: An autonomous reliability-aware negotiation strategy for cloud computing environments. In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on* IEEE, 2012, S. 284–291 (cited on page 5).

Dastjerdi u. Buyya 2015

Dastjerdi, Amir V. ; Buyya, Rajkumar: An Autonomous Time-Dependent SLA Negoti-

ation Strategy for Cloud Computing. In: *Comput. J.* 58 (2015), S. 3202–3216 (cited on pages 2, 3 and 5).

Feng et al. 2014

Feng, Yuan ; Li, Baochun ; Li, Bo: Price competition in an oligopoly market with multiple iaas cloud providers. In: *IEEE Transactions on Computers* 63 (2014), Nr. 1, S. 59–73 (cited on page 3).

Fill u. Härer 2018

Fill, Hans-Georg ; Härer, Felix: Knowledge Blockchains: Applying Blockchain Technologies to Enterprise Modeling. In: *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018 (cited on page 29).

Hamari et al. 2016

Hamari, Juho ; Sjöklint, Mimmi ; Ukkonen, Antti: The sharing economy: Why people participate in collaborative consumption. In: *Journal of the association for information science and technology* 67 (2016), Nr. 9, S. 2047–2059 (cited on page 3).

Haq 2010

Haq, Irfan U.: *A Framework for SLA-Centric Service-Based Utility Computing*. `http://eprints.cs.univie.ac.at/4165/`. Version: 2010 (cited on pages 14 and 15).

Lamanna et al. 2003

Lamanna, D D. ; Skene, James ; Emmerich, Wolfgang: Slang: A language for defining service level agreements. In: *NINTH IEEE WORKSHOP ON FUTURE TRENDS OF DISTRIBUTED COMPUTING SYSTEMS, PROCEEDINGS* IEEE COMPUTER SOC, 2003, S. 100–106 (cited on page 15).

Liu et al. 2011

Liu, Fang ; Tong, Jin ; Mao, Jian ; Bohn, Robert ; Messina, John ; Badger, Lee ; Leaf, Dawn: NIST cloud computing reference architecture. In: *NIST special publication* 500 (2011), Nr. 2011, S. 1–28 (cited on page 53).

Merkle 1980

Merkle, Ralph C.: Protocols for Public Key Cryptosystems. In: *1980 IEEE Symposium on Security and Privacy* (1980), S. 122–122 (cited on page 12).

Nakamoto 2008

Nakamoto, Satoshi: Bitcoin: A peer-to-peer electronic cash system. (2008) (cited on pages 3, 4 and 9).

Nepal et al. 2008

Nepal, Surya ; Zic, John ; Chen, Shiping: WSLA+: Web service level agreement language for collaborations. In: *Services Computing, 2008. SCC'08. IEEE International Conference on* Bd. 2 IEEE, 2008, S. 485–488 (cited on page 15).

Opara-Martins et al. 2014

Opara-Martins, Justice ; Sahandi, Reza ; Tian, Feng: Critical review of vendor lock-in and its impact on adoption of cloud computing. In: *Information Society (i-Society), 2014 International Conference on* IEEE, 2014, S. 92–97 (cited on page 1).

Pittl et al. 2018

Pittl, Benedikt ; Werner, Mach ; Schikuta, Erch: *Bazaar-Blockchain: A Blockchain for Bazaar-based Cloud Markets.* 2018 (cited on pages 1, 5 and 29).

Poon u. Buterin 2017

Poon, Joseph ; Buterin, Vitalik: *Plasma: Scalable Autonomous Smart Contracts.* 2017 (cited on pages 56 and 57).

Poon u. Dryja 2016

Poon, Joseph ; Dryja, Thaddeus: *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments.* 2016 (cited on page 56).

Ranaweera u. Prabhu 2003a

Ranaweera, Chatura ; Prabhu, Jaideep: The influence of satisfaction, trust and switching barriers on customer retention in a continuous purchasing setting. In: *International journal of service industry management* 14 (2003), Nr. 4, S. 374–395 (cited on page 3).

Ranaweera u. Prabhu 2003b

Ranaweera, Chatura ; Prabhu, Jaideep: On the relative importance of customer satisfaction and trust as determinants of customer retention and positive word of mouth. In: *Journal of Targeting, Measurement and Analysis for Marketing* 12 (2003), Jul, Nr. 1, 82–90. `http://dx.doi.org/10.1057/palgrave.jt.5740100`. – DOI 10.1057/palgrave.jt.5740100. – ISSN 1479–1862 (cited on page 3).

Seijas et al. 2016

Seijas, Pablo L. ; Thompson, Simon J. ; McAdams, Darryl: Scripting smart contracts for distributed ledger technology. In: *IACR Cryptology ePrint Archive* 2016 (2016), S. 1156 (cited on page 16).

Serguei 2018

Serguei, Popov: *The Tangle v1.4.3.* Website, April 2018 (cited on page 12).

Takabi et al. 2010

Takabi, H. ; Joshi, J. B. D. ; Ahn, G.: Security and Privacy Challenges in Cloud Computing Environments. In: *IEEE Security Privacy* 8 (2010), Nov, Nr. 6, S. 24–31. http://dx.doi.org/10.1109/MSP.2010.186. – DOI 10.1109/MSP.2010.186. – ISSN 1540–7993 (cited on page 2).

Vitalik u. Virgil 2017

Vitalik, Buterin ; Virgil, Griffith: *Casper the Friendly Finality Gadget*. 2017 (cited on pages 8 and 17).

Waeldrich et al. 2011

Waeldrich, Oliver ; Battré, Dominic ; Brazier, Francis ; Clark, Kassidy ; Oey, Michel ; Papaspyrou, Alexander ; Wieder, Philipp ; Ziegler, Wolfgang: WS-agreement negotiation version 1.0. In: *Open Grid Forum* Bd. 35, 2011, S. 41 (cited on pages 2, 15, 19, 30, 33 and 34).

Werner 2017

Werner, Mach: *A Simulation Environment for WS-Agreement Negotiation Compliant Strategies*. 2017 (cited on pages 33 and 34).

Wood 2017

Wood, Gavin: Ethereum: a Secure Decentralised Generalised Transaction Ledger, 2017 (cited on pages 5, 11, 29 and 31).

Thesis Affirmation

I affirm that this thesis was written by myself without any unauthorised third-party support. All used references and resources are clearly indicated. All quotes and citations are properly referenced. This thesis was never presented in the past in the same or similar form to any examination board.

Vienna, the September 25, 2018