University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

SLAMer: a blockchain-based SLA Management System

Carlos Schweizer
Zürich, Switzerland
Student ID: 15-702-764

Supervisor: Eder John Scheid, Cristian Killer
Date of Submission: September 1, 2019

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Seit Bitcoin im Jahr 2009 das Blockchain-Konzept als vertrauenswürdige Umgebung für den Austausch von Geldmittel zwischen unbekannten Peers eingeführt hat, erregte dieses Konzept grosse Aufmerksamkeit aufgrund der Möglichkeiten, Probleme anzugehen, die mit Vertrauen außerhalb des Finanzbereichs verbunden sind. Ein Bereich, der in der Vertrauensbeziehung zwischen unbekannten Peers existiert, ist die Kompensation von Service Level Agreements (SLA), insbesondere die Durchsetzung der in SLAs festgelegten Bedingungen, wenn ein Verstoß festgestellt wird. Dabei ist der Service Provider (SP) stärker positioniert als der Kunde, da er Forderungen des Kunden im Bezug auf Rückerstattung überprüft und entscheidet, ob er den Kunden entschädigt oder nicht. Daher ist die Aufgabe, einen Verstoß zu beweisen, eine Herausforderung denn der Kunde muss darauf vertrauen, dass der SP ehrlich handelt. Es wurden Forschungsarbeiten zur Bereitstellung datenbank- und blockchainbasierter Lösungen für diese Probleme durchgeführt, wobei letztere mehr Vorteile bieten, wie z.B. die Durchsetzung von Zahlungen und die unveränderliche Datenspeicherung. So wurde in dieser Arbeit ein blockchainbasiertes SLA-Managementsystem, genannt `SLAMer`, entwickelt, um bei der Verwaltung des SLA-Lebenszyklus unter Verwendung von blockchainbasierten Smart Contracts (SC) zu helfen, um Zahlungen (z.B. Servicegebühr und Kompensation) beider Parteien durchzusetzen und die SLA-Bedingungen unveränderlich zu speichern und so die Integrität des Prozesses zu gewährleisten. `SLAMer` wurde nach der WSLA-Sprache konzipiert und ein Proof-of-Concept (PoC) implementiert. Der PoC ermöglicht es einem externen Monitoring Service, Messungen der einzelnen Services zu senden, die wiederum im SC verifiziert werden. Der `SLAMer` PoC wurde unter wirtschaftlichen, Management-, Performance- und Usability-Gesichtspunkten bewertet. Diese Bewertung zeigt, dass dieser Ansatz in der Lage ist, die oben genannten Herausforderungen von SLAs zu bewältigen. Die Durchsetzung von Entschädigungen des SP an den Kunden wird durch die SCs gewährleistet, da im Vertrag eingeschlossene Kryptowährungen bei bestimmten Ereignissen an die entsprechende Partei abgetreten werden. Die Kosten für die Überprüfung der Serviceparameter in einem SC sind jedoch immer noch beträchtlich hoch. Darüber hinaus sind die Leistung und Benutzerfreundlichkeit von `SLAMer` eng mit der Blockchainleistung gekoppelt, was zu einer Latenzzeit in Bezug auf die Reaktionszeit von Interaktionen mit einem SC führt. Trotz dieser Herausforderungen zeigte die Bewertung, dass der Ansatz machbar ist und die Lösung in der Lage ist, Teile des SLA-Lebenszyklus zu bewältigen und so die manuelle Komplexität und Interaktion zu reduzieren.

Since Bitcoin introduced the blockchain concept in 2009 as a trusted environment to exchange funds between unknown peers, this concept gained a lot of attention due to the possibilities to address problems that involve trust outside the financial area. One

area that exists in the trust relationship between unknown peers is the compensation of Service Level Agreements (SLA), especially, the enforcement of the terms specified in SLAs when a violation is detected. In this process, the Service Provider (SP) is in a stronger position than the customer, since the SP verifies a claim made by the customer and decides whether or not to compensate the customer. Thus, the task of proving a violation is challenging because the customer must trust that the SP will act honestly. Research has been made in providing database-based and blockchain-based solutions to these problems with the latter presenting more benefits, such as payment enforcement and immutable data storage. Thus, in this thesis, a blockchain-based SLA management system, called `SLAMer`, was designed to aid in the management of the SLA lifecycle using blockchain-based Smart Contracts (SC) to enforce payments (e.g., service fee and compensation) by both parties and store the SLA terms in an immutable manner, providing integrity to the process. `SLAMer` was designed following the WSLA language and a Proof-of-Concept (PoC) was implemented. The PoC allows an external monitoring service to send measurements of the services which in turn are verified in the SC. `SLAMer` PoC was evaluated in terms of economic, management, performance and usability aspects. This evaluation shows that this approach is able to address the aforementioned challenges of SLAs. Enforcement of compensations from the SP to the customer is guaranteed by the SCs, as cryptocurrencies locked inside the contract are relieved to the corresponding party on certain events. However, the cost of verifying service parameters inside a SC are still considerably high. Further, the performance and usability of `SLAMer` are tightly coupled to the blockchain performance; therefore leading to latency in regard to response time of interactions with a SC. Despite these challenges, the evaluation showed that the approach is feasible and `SLAMer` is capable of addressing parts of the SLA lifecycle, aiding to reduce manual complexity and interaction.

Acknowledgments

I want to thank Eder Scheid for being an excellent supervisor and supporting me with writing this thesis. His advises helped me tremendously in structuring and planning, as well as in leading me in the right direction when it came to problems in understanding.

Contents

Chapter 1

Introduction

Service Level Agreements (SLA) are legal contracts between Service Providers (SP) and customers. In these SLAs, service requirements, such as performance expectations, and service details (*e.g.,* price and service validity) are negotiated and defined. Performance expectations are defined as Quality-of-Service (QoS) requirements, and detail, for instance, that the availability of the service (*i.e.,* server uptime) should be more than 99.99%. Further, QoS requirements include data throughput or even response times to phone calls. If one of these QoS requirement is violated (*i.e.,* not met or delivered), the SP has to compensate the customer according to agreed terms, which are also defined in the SLA. However, until now, the whole process of managing SLAs and compensations consists of various manual steps that are time consuming and prone to errors.

Moreover, the nature of SLAs brings three main issues that challenges the current situation. First, the verification of an SLA has to be performed manually. Many tools exist for service monitoring to detect any violations [1] but the main challenge lies in the enforcement of such agreements. Second, the SP is in a stronger position than the customer due to the fact that he is authorized to verify the violation and is able to decide whether or not to compensate the customer. This leads to the third challenge, the process of proving the violation. The customer has to convince the SP that a violation has happened.

A possible solution to these problems could be to rely on the blockchain [2] and Smart Contracts (SC) [3]. The concepts of blockchain and SCs have gained an immense amount of interest, not only in the area of cryptocurrencies but in all areas where information technology is present. Where problems arised in the past, blockchain tries to find its place by providing new solutions.

Blockchain technology can help by removing the need for a Trusted-Third-Party (TTP) to verify transactions and provide an immutable data storage, *i.e.,* once the data is appended in the blockchain it cannot be removed. SCs might help addressing the aforementioned problems due to their property of automatic execution upon a transaction, immutable source code, and output verified by the whole blockchain.

1.1 Description of Work

Therefore, in this thesis, the topic of SLAs in connection with blockchain is investigated. Some existing works addressing the traditional SLA management are provided, followed by research of SLAs in relation to SCs. The focus lies on the coverage of the different SLA management lifecycle steps in each work. A prototype of a blockchain-based SLA Management system, called `SLAMer`, is designed and developed. This SLA management system allows service providers and customers to manage their SLAs in terms of creation, review, deployment and most important, penalty enforcement. `SLAMer` stores all the relevant data in a SC which is deployed on the Ethereum network. This SC takes care of the verification of the defined Service Level Objectives (SLO) and is responsible for automatically transferring funds to the correct parties. `SLAMer` is composed of a Graphical User Interface (GUI) for user interaction, a back-end containing the business logic (*i.e.,* SLA lifecycle) and a blockchain connector to deploy and interact with the SC.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 provides the theoretical background of the thesis. It explains the underlying concepts of SLAs, SLA lifecycle, the blockchain technology, and SCs. Chapter 3 provides related works to the thesis by describing existing approaches regarding SLA Management, and blockchain-based SLA management. In Chapter 4, the `SLAMer`'s architecture, design and SLA template are explained. It is followed by implementation details including the user interface, its functions, and the Ethereum SC that contains the SLA information. In Chapter 5, the design and implementation are evaluated against the thesis' goals and requirements. For this purpose, economic and management aspects are considered as well as usability and performance. Lastly, Chapter 6 concludes the thesis with a summary and future work.

Chapter 2

Background

This chapter presents the theoretical background necessary to understand the approach presented in the thesis. First, the concept of Service Level Agreements (SLA) and its lifecycle are presented in Section 2.1. Then, in Section 2.2, the blockchain technology is described. Section 2.3 explains the Ethereum platform as well as Ethereum-based Smart Contracts (SC) in Section 2.3.1. Finally, Section 2.3.2 introduces the basic concepts and features of the Solidity language.

2.1 Service Level Agreements (SLA)

Assuming a cloud scenario where a customer requires one or more virtualized services, *e.g.,* storage or hardware, he/she has to search for a SP that is able to meet the specific demands for these services. After the customer has identified the SP, both must negotiate which services will be delivered and what are the requirements for such services. Then, they commit to an agreement, which is often referred to as an SLA, that describes the expected service level that the SP must provide the customer and ensures that QoS metrics are met [4]. Each SLA between an customer and a SP can contain several SLOs, which are the concrete definition with measurable values of QoS metrics. For instance, taking Figure 2.1 as an example, the availability of a Web Server is measured by the SLO defining the QoS metric "`uptime` $\geq$ `99.9999%`", and the performance is measured by the SLOs defining the QoS metrics "`throughput` $\geq$ `200 kB/s`" and "`response time` $\leq$ `10 ms`".

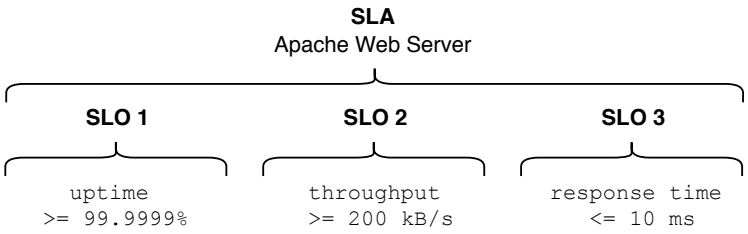

Figure 2.1: Service Level Objectives (SLO) Examples

Monthly Uptime Percentage (X)	Compensation		
	Azure	**Amazon**	**Google**
$99\% \leq X < 99.99\%$	10%	10%	10%
$95\% \leq X < 99\%$	25%	30%	25%
$X < 95\%$	100%	100%	50%

Table 2.1: Compensation in Service Credits

Moreover, SLAs define the penalties that a SP is obligated to pay to the customer SLA if they fail to deliver the expected QoS. The compensation (*i.e.,* penalty) that an SP must pay for the customer if it did not achieve the agreed SLOs vary from SP to SP. For example, in Table 2.1 different compensation values are presented for the same SLA from Azure [5], Amazon [6] and Google [7], where X represents the measured percentage value. All three providers ensure to credit 10% of the service bill to the customer if X is in the range of 99% and 99.99%. On one hand, Azure and Google guarantee to compensate the customer 25% of service credits if the service uptime should be less than 99% but higher or equal to 95%. Amazon, on the other hand, gives back 30% to the customer for the same range. If the total uptime results in being less than 95%, Azure and Amazon give a full refund (*i.e.,* 100% is paid back). In contrast, Google only offers a refund of 50%. The compensation process is manual at the moment; this means that the customer is responsible for claiming an SLA violation and filling the report in order to be compensated.

2.1.1 SLA Lifecycle

SLAs are subject to a lifecycle which every new SLA has to go through. There are different definitions on the terminology of the SLA lifecycle [8]. However, it essentially bases on 6 lifecycle phases, which are depicted in Figure 2.2. The phases, definitions, and terminology used on this thesis are based on [9] and described below.

1. **Discover Service Provider** - The customer has to identify a SP that provides the desired resources and services. The identification of the SP depends on requirements from the customer, *e.g.,* defined budget, Geo-location, security mechanisms, privacy concerns.

2. **Define SLA** - Once the customer found a SP, usually a negotiation takes place where both parties try to reach a consensus regarding the SLOs. Also, prices and penalties are defined in this phase. It is crucial that both parties have the same understanding of each other's expectations so that the SLA is unambiguous and no problems arise after the SLA becomes effective. If the involved parties do not come to an agreement, the lifecycle process starts over until a consensus is reached.

3. **Establish Agreement** - This phase consists of defining and developing the template in which the SLA will take place. Both parties are obliged to the terms defined in the previous phase as soon as they sign the contract. The defined services are deployed and ready to be accessed and utilized by the customer.

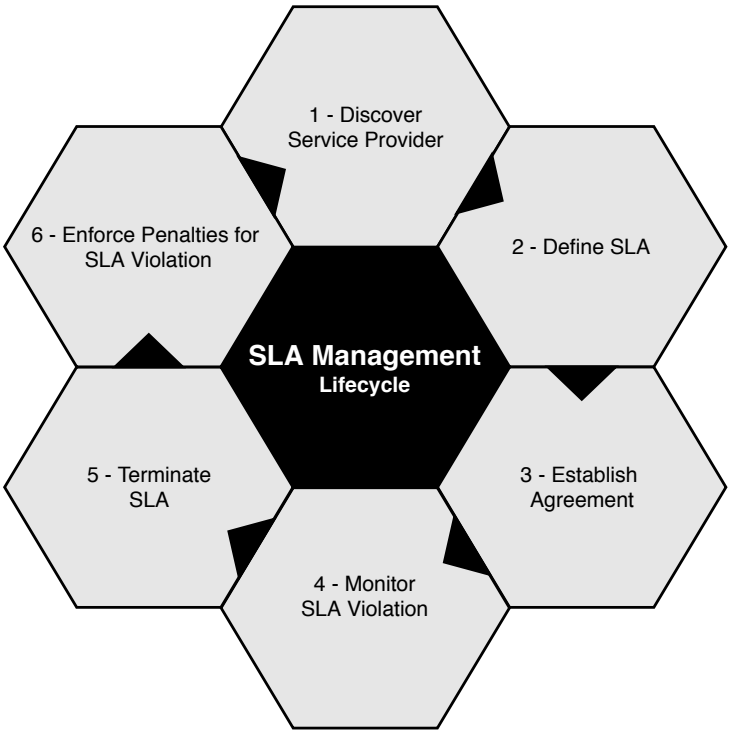

Figure 2.2: SLA Lifecycle Example [9]

4. **Monitor SLA Violation** - The deployed services have to be monitored to ensure that the SP meets the agreed terms. Both parties monitor the services on their own or rely on a third party solution. In either case, the monitoring solution has to provide reliable measurements.

5. **Terminate SLA** - If no violation occurred and the service was delivered as agreed the SLA terminates when it expires. Depending on the terms defined in Phase #2, it can also terminate earlier. For example, if the number of detected violations was above a defined threshold.

6. **Enforce Penalties for SLA Violation** - In the case where the SP does not meet the specified performance levels, the customer needs to be compensated. The compensation value is calculated according to the penalties defined in the SLA. These compensation can be in form of fiat-currency or service-credits. For example, Amazon compensates customers in the form of service-credits within one billing month after the violation has been confirmed by the responsible team [6].

2.2 Blockchain

Blockchain as a real world implementation was first introduced with the creation of Bitcoin in 2009 and gained attention later with the speculation on the value of Bitcoins [2]. A blockchain is a distributed ledger managed by a peer-to-peer network in which records are stored as transactions in a block. For every transaction, the transaction data is signed and verified by other participants in the network using cryptographic algorithms. If a majority

of participants agree that the transaction is valid, a new block is added to the blockchain and shared to all other nodes [10]. To include a new block in the blockchain, a node, called miner, has to find a solution for a cryptography puzzle, which is computational expensive to calculate, but easy to verify. Once a miner finds the answer to the puzzle, it shares it with the other nodes who verify if it is correct and then append the block to the blockchain.

Each block uses a hash in order to point to the previous block's header as shown in Figure 2.3. Thus, when following all this pointers, one will find the first block in the chain, called the genesis block. In order to alter information in a block, all following block hashes would need to be recomputed, since any change to a block changes its hash. The computational effort to perform this operation becomes exponentially high, which leads to the fact that data can be stored in the blockchain but not deleted or altered, *i.e.,* the data is immutable once it is appended in the chain.

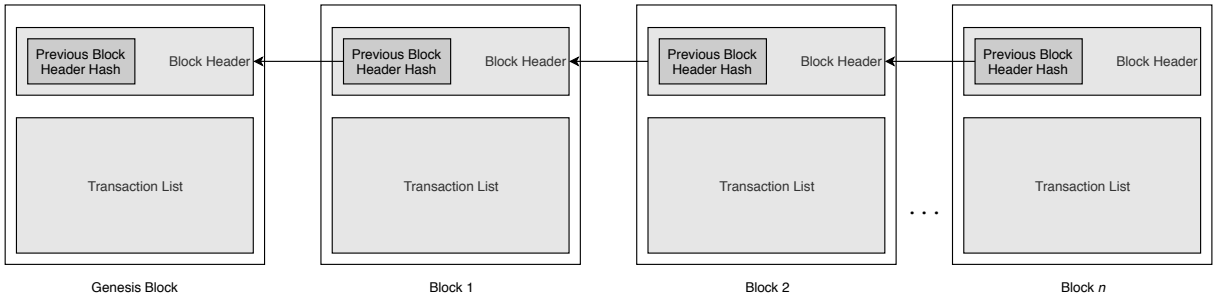

Figure 2.3: Blockchain Example

2.3 Ethereum

The Ethereum platform is a public account-based ledger. It provides a decentralized virtual machine (VM) that serves as a run-time environment for SCs, known as the Ethereum Virtual Machine (EVM). As it is outlined in the Ethereum yellow paper, Ethereum as a whole "can be viewed as a transaction-based state machine" [11]. It starts with a genesis state which transitions to another state when transactions are executed. Such transactions are stored in blocks, as explained in Section 2.2. In addition to the transactions, the block also stores an identifier of the current state.

Figure 2.4 depicts a simplified illustration of this state-transition. The blockchain is in some state t with two accounts A and B, having a balance of 20 and 25 ETH respectively. When executing a transaction *e.g.,* transferring 10 from account B to account A, both accounts end up in a new state. Thus, the blockchain reaches some new state *t+1.*

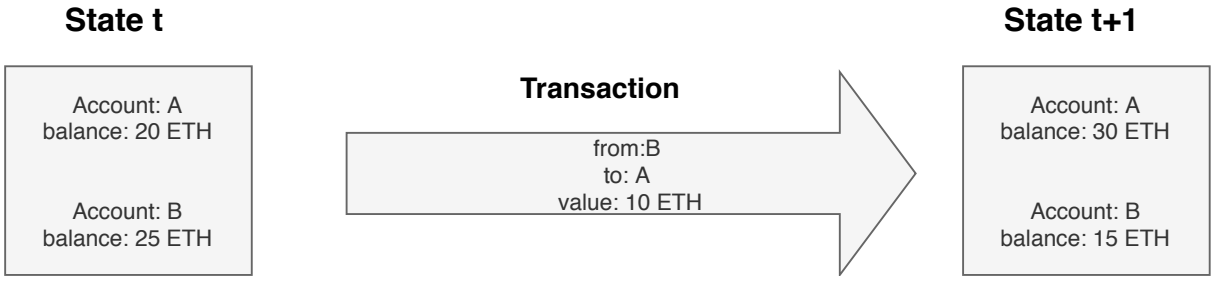

Figure 2.4: State transition from one state to the next

In order for this state-transition to happen, nodes in the network need to mine the blocks containing these transactions which is incentivized by the intrinsic currency Ether. The smallest subdenomination of Ether is called Wei, in which all values are stored. One Ether corresponds to 10^{18} Wei.

2.3.1 Ethereum-based Smart Contracts

SCs are computer code that map contracts or technically support the negotiation or execution of a contract. This means, contracts are not enforced by law but by hardware and software [12]. As Nick Szabo [12] states in his paper, the main goal is to "reduce mental and computational transaction costs imposed by either principals, third parties, or their tools". These SC are executed by the consensus mechanism of the corresponding blockchain. The content of SCs is, in other words, computer code that implements a certain logic, depending on a contract's purpose. SCs can be implemented in many blockchains, such as in Bitcoin or Cardano [13], although they are limited in terms of functionality. Ethereum provides an advanced environment for creating and running Turing-complete SCs.

The source code of SCs is developed outside of the blockchain itself, usually using a dedicated Integrated Development Environment (IDE) *e.g.,* Remix in case of using Solidity [14]. Once the SC is ready for deployment, developers wrap the contract code in a transaction and send the transaction with no recipient to the blockchain. In the blockchain, the transaction data becomes an executable program in the EVM. Besides the executable code, the SC has a contract address, a balance and a state [11]. It is important to note that the data stored in a contract is publicly visible to external observers. Thus, sensitive data should not be stored in a SC without encryption.

Since the blockchain is immutable, the SC code is immutable as well. That means, the source code can never be changed again once it is appended to the blockchain. Once a SC is deployed, its code will only be executed if transactions are sent to the SC address. Because miners verify all the transactions made to the blockchain, the SC is run by them whenever a call to a contract function happens. The output of such interactions can be trusted because SCs are deterministic. This determinism is required, so that the network nodes execute the SC and get the same result, and thus, can reach consensus.

Due to the fact that Ethereum's bytecode is Turing-complete, fees are charged for every computation so that issues of network abuse, such as Denial-of-Service (DoS) attacks, are

avoided [11]. These fees are paid in units called gas. Thus, a SC function can only be successfully called if a transaction contains enough gas for its execution. Gas itself does not exist outside of the transaction; it is purchased at a certain gas price in Ether at the point of execution. The gas price can be chosen freely, but the higher the gas price is set, the faster the transaction is included by miners, since the miners will prioritize higher gas prices to receive more fees.

2.3.2 Solidity

Usually, SCs are written in a high-level language. Different languages exist, such as Serpent [15], Vyper [16] and LLL (Lisp-like language)[17], but the most used one is Solidity [18]. Solidity is a C++-like language with static typing that also resembles JavaScript and Python. It also supports inheritance and polymorphism and lets SCs be structured as *contracts* in a class-like manner, as it is common in object-oriented languages like Java. Despite the differences between these languages, each of them is compiled to a series of bytecode instructions which the EVM can natively execute. In the following items, key features of Solidity are presented [18].

- **Common Data Properties**: Solidity supports multiple data types as it is common in other languages, such as string, integer, enum and boolean. However, floating point numbers are not supported; thus, every number is represented as an integer or unsigned integer (`uint` *i.e.,* non-negative numbers). In addition, there exists the `address` type to access members such as the balance.
- **Struct**: Structs are similar to classes, allowing the definition of new types by grouping different variables as well as other structs.
- **Events**: Events are a convenient way of notifying external applications about a SCs state changes. For example, clients can listen for specific events and react accordingly as soon as they are emitted. Moreover, they are useful for logging purposes, so that additional information about the associated SCs and their transactions can be retrieved.
- **Function Modifiers**: They act as an extension to functions which are executed before a function runs. This is mostly used to evaluate conditions, for instance access checks. For example, there might be operations that only specific accounts are allowed to perform and must therefore be checked if a caller is permitted to execute a function. This check is performed in the modifier which can be applied to multiple functions, removing these conditions from the function body itself. Listing 2.1 presents a simple function modifier. The "_" at line 3 serves as a placeholder for the code of the function the modifier is applied to *e.g.,* the `withdraw` function.
- **Special Variables**: Solidity also provides special variables that one can always access inside a SC, the most important ones being `msg`, `block` and `tx`. These allow access to information such as the caller of the contract, the current block number or the current transaction. For instance, in Listing 2.1 on line 2, the sender of the message is accessed to control the access to the function.

```
1       modifier onlyCustomer() {
2           require(msg.sender == deployedSLA.customer, "Only the
                customer can call this function");
3           _;
4       }
5
6       function withdraw() public onlyCustomer {
7           // function logic
8       }
```

Listing 2.1: Modifier restricting access only to the customer

Chapter 3

Related Work

This chapter presents the works that are related to the core topics of this thesis. First, in Section 3.1, an outlook on traditional SLA management approaches is presented. Then, in Section 3.2, blockchain-based SLA management solutions are described. Finally, this chapter ends with a discussion on the shortcomings of such works in Section 3.3.

3.1 Traditional SLA Management Approaches

Any works describing scientific approaches to SLA management which are not just under research anymore is herein referred to as "traditional". This also includes investigation and proposals for SLAs in cloud-computing. In the early 2000's, when SLAs started to become widely adopted, the IBM Research Division and the IBM Software Group developed the WSLA language [19]. An important aspect is the development of a type system for describing SLAs. WSLA divides a SLA in three parts: parties, service description, and obligations. Its flexible design allows to describe a wide variety of parameters.

Based on the WSLA research, [20] describes a framework for providing differentiated levels of service through the use of automated management and SLAs. The authors propose a "Web-service contracting" environment dedicated to interactions between customer and SPs in terms of service offering by SPs and service subscriptions of customers. Further, a "Web-services-on-demand" environment exists, where relationships between customers and SPs are formed dynamically, and the SPs resources are provisioned on demand. The "Web-service provisioning" environment takes care of resource allocation and the provisioning of the services, performed in a specific workflow.

In [21], the authors propose a SLA management system which consists of a pre-runtime and a runtime environment. The former focuses on registration and service search. SPs can register and publish their services so that potential customers can search for these services which match their needs. The latter is responsible for monitoring and controlling the service runtime states. This also includes making a punishment decision in case of a SLA violation. The authors cover every phase except the last, the penalty enforcement,

which is not explicitly addressed, as the paper does not define how compensations are performed in case of a violation.

In [22], the authors propose a model that relates to the first (*i.e.,* **Discover Service Provider**), second (*i.e.,* **Define SLA**) and the fourth (*i.e.,* **Monitor SLA Violation**) phase of the SLA management lifecycle. They introduce a regulator that is "involved in the monitoring framework process by launching audits, policy making, and QoS monitoring" [22]. Further, the regulator is able to help the customer in the identification and ranking of SPs by considering the results of previous monitoring of SLAs .

3.2 Blockchain-based SLA Management

At the current state, the use of SCs for SLA Management is still under research and different authors have proposed frameworks. In [23] is proposed the use of the Ethereum Blockchain for hosting the SCs and enforcing payments between the parties in the Network Function Virtualization (NFV) context. [24] went for a similar direction in terms of using the Ethereum Blockchain. They focus on the specification through the Resource Description Framework (RDF) to formally describe Web APIs and their SLAs. However, the compensation of the involved parties is not described. In contrast, [25] focus on the enforcement of those SLAs. They propose a Witness Model which consists of a *Witness Committee* that is in charge of monitoring the service on which the parties agreed on. On violation detection, they reach a consensus on whether or not a violation happened, and if so, they report it to the SC to execute a payment.

The authors of [26] focus on the usage of Ethereum SCs in the context of Small-Cell-as-a-Service (SCaaS) agreements between network operators and small-cell owners. They did, however, not focus on the monitoring and or compensation part. Moreover, [27] proposed a framework to cover all the phases of the SLA lifecycle. They suggest an architecture with two different networks: the side-chain and blockchain network. The former is used for heavy computations such as discovery and negotiation of SLAs, the latter is used for SC execution. An Oracle serves as an interface between them. The detection of SLA violations is done in a similar way as [25], since they propose the inclusion of network participants to take over the role of an *auditor*.

3.3 Discussion

As it is outlined in Section 3.1, there has been some research since about two decades trying to facilitate the SLA management. The WSLA language developed by IBM was quite influential and served as a foundation for later research. Some of the proposed solutions present dedicated environments for the different lifecycle phases. Other authors suggest an integration of a regulator to support the monitoring phase. This regulator also provides an objective reputation system by collecting data from previous SLAs and ranking the SPs based on that. Despite the proposals being sophisticated, they all lack in addressing the enforcement of penalties. Additionally, the introduction of the aforementioned regulator

means that there is another layer of trust, because the customer must trust that the regulator is not SP-biased.

In Table 3.1 an overview is presented of the related work and the life-cycle phases that are addressed. In such a table the symbol "✓" means fully addressed, "✓✗" represents partially fulfilled, and "✗" means not addressed.

	SLA Life-cycle Phases					
Work	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**
[23]	✗	✗	✗	✓✗	✓	✓
[24]	✗	✗	✗✓	✗	✗✓	✗
[25]	✗	✗	✓	✓	✓	✓
[26]	✗	✗	✓✗	✗	✗	✓✗
[27]	✓	✓	✓	✓	✓	✓
This Work	✗	✓✗	✓✗	✓✗	✓	✓

Table 3.1: Related Work and Addressed Phases

The author of [23] takes a focused approach and addresses the enforcement of payments. Monitoring is partially covered as only the validation of measurements takes place in the SC. [24] is similar, but does not cover the customer compensation. While [26] only partially covers some phases, [25] addresses phases #3 to #6, leaving out the discovering and definition phase. [27] though, covers the whole SLA lifecycle and seems to be quite sophisticated. This work focuses on the phases #5 and #6 and only partially addresses phases #2 to #4. The definition phase happens outside of `SLAMer` but in order to guarantee a valid SLA, a review process takes place that mimics this phase. The establishment corresponds to the SLA being ready for deployment and activated. Monitoring the services is performed by a third party solution, however the verification of the service levels takes place inside the SCs.

Overall, research in the field of SLAs in conjunction with SCs tends to address the second half of the SLA lifecycle. Mainly, the availability of a monitoring service is important so that SCs can verify the measurements and take payments accordingly. The first half of the SLA lifecycle has been widely covered in previous research and is not of big interest regarding the integration of SCs.

Chapter 4

Blockchain-based SLA Management System

In this chapter, `SLAMer` (**SLA** **M**anag**er**) is presented. `SLAMer` relies on the blockchain to provide a trusted and immutable SLA management solution. SPs can register their SLA by collecting all the technical details specified in the agreement. After the customer agrees with the data entered by the SP, the SLA is deployed as a SC and is responsible for verifying the services in scope. By this mechanism, the problems of distrust can be addressed by automating the process of paying a compensation to the customer and proving immutability to the data stored in the SC.

4.1 `SLAMer` Design

The architecture of `SLAMer` is depicted in Figure 4.1. Both the SP and customer can access `SLAMer` by logging in via the *GUI* which serves as the frontend. The roles are not mutually exclusive. A party can be in either roles, depending on the SLA, *i.e.,* a party can be a SP in one SLA and a customer in another. Any actions performed by the parties are sent to the backend through the *SLAMer API* and then processed by the *SLA Manager* and the *Blockchain Connector*. The components that compose `SLAMer` are described in the next items.

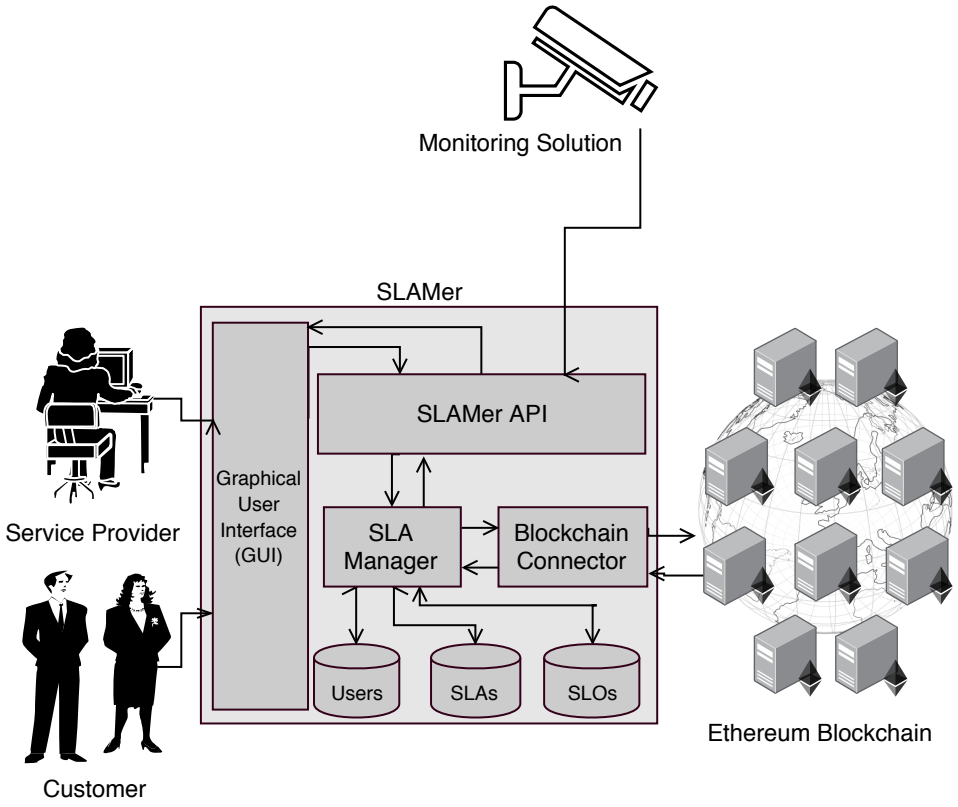

Figure 4.1: `SLAMer` Architecture

- **Graphical User Interface**: *ServiceProvider* and *Customer* can register themselves over the GUI and log in to see their SLAs and the state they are currently in.

- **SLAMer API**: The REST API endpoints are implemented here. Any HTTP requests (*e.g.,* POST and GET) coming from the frontend are handled and passed to the *SLA Manager*. The API is also accessible for *Monitoring Solutions*. These are services which actually measure the parameters defined in an SLA. These measurements are sent to the *SLAMer API* and passed to the corresponding SC.

- **SLA Manager** is responsible for the general business logic. This is the core component of `SLAMer` and takes care of the maintenance of the SLAs and security. Furthermore, the *SLA Manager* is responsible for user registration, authentication and authorization. Furthermore, it is tightly coupled to the *Blockchain Connector* to keep the SLA states in sync with the SCs.

- **Blockchain Connector** is in charge of deploying SCs and fetching information from them. The *Blockchain Connector* also listens for events emitted by SCs on the blockchain, such as state changes.

- **Database**: The database stores all the SLA information. It persists all the relations to involved parties (*i.e.,* users) and the specific SLO parameters. For each party, their wallet details *i.e.,* public and private key, are stored to sign transactions. Also, the SC address on the blockchain is stored along with the SLA.

4.2 SLA Definition

This Section first describes the WSLA definition and provides an overview about its concepts. Second, Section 4.2.2 explains how the SLA template for `SLAMer` was derived from the WSLA definition.

4.2.1 WSLA SLA Template/Definition

For the foundation of the data model, the WSLA framework developed by IBM [19] was taken. This framework provides an XML-based representation of an SLA, allowing users to register SP and customers as well as an arbitrary number of supporting parties. Figure 4.2 depicts the most important object types as described in their document [19].

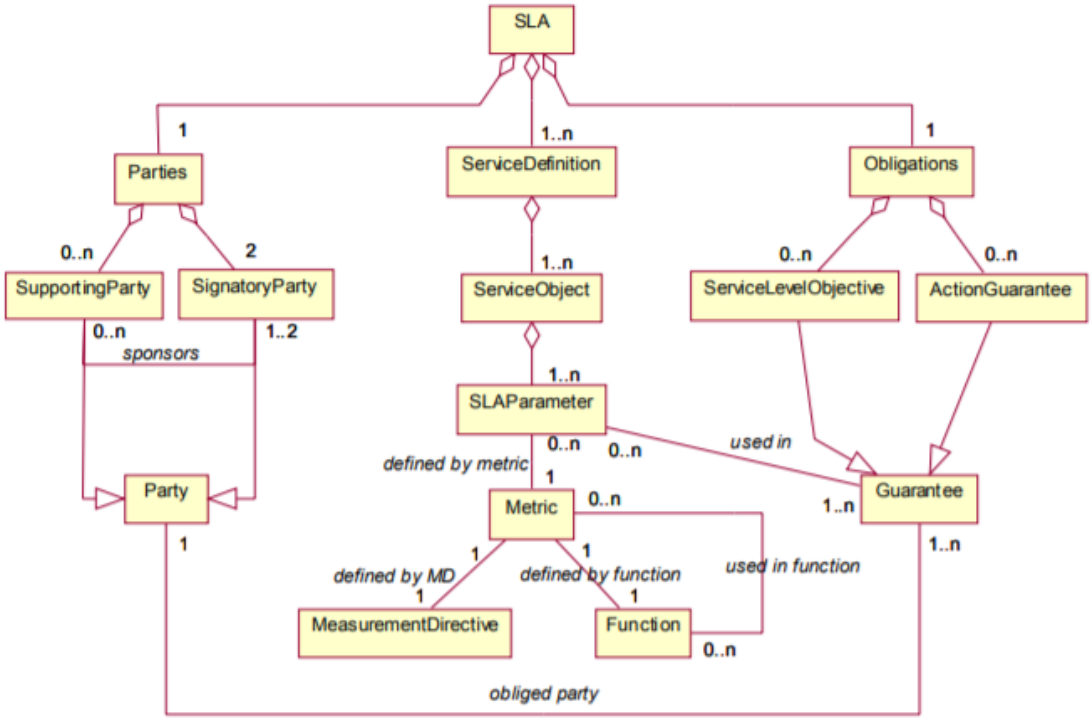

Figure 4.2: UML class diagram of the conceptual object types [19]

Conceptually, an SLA can be divided into three sections: the first section describes the involved parties, the second specifies one or more service definitions and the third section defines the parties' obligations.

In the parties' section, SP and service customer are aggregated as *Signatory Party*. *Supporting Party* denotes the many different services provided by a signatory party to provide measurements and condition evaluation services.

ServiceDefinition contains all the services represented as *ServiceObject* which is an abstract representation of a service. Its relevant properties are stored as *SLAParameter* and are combined with a *Metric*. A *Metric* is either a *MeasurementDirective* (*i.e.,* how a value is measured) or a *Function* (*i.e.,* how a metric is computed).

The last section defines the *Obligations* in an SLA. There are two types of obligations, the first being *ServiceLevelObjective* and the second being *ActionGuarantee*. Essentially, a SLO represents a guarantee that a *SLAParameter* will be in a given state for a defined time period. On the one hand, SLOs are normally obligations of the SP who has to guarantee that his provided services comply to given QoS level. On the other hand, an action guarantee describes a specific action that has to be performed in a defined situation. This guarantee can have any party as the obliged. To give an illustration, an *ActionGuarantee* can be a payment of the SP to the customer in case the SP fails to deliver the agreed QoS.

4.2.2 `SLAMer` SLA Template

The main focus of this Bachelor thesis is to research the management of SLAs using blockchain-based SCs to provide integrity and trust to parties involved in the SLA lifecycle. Therefore, for the sake of simplicity and due to time constraints, the WSLA definition was refined to include the main parameters of an SLA. However, it is possible and planned, as future work, to include the complete WSLA template in the implementation of `SLAMer`. This refinement aid in the development of the whole design and in the implementation of the `SLAMer` SC, which is described in Section 4.6.2.

Figure 4.3 shows the simplified version of the diagram presented in Figure 4.2. The refinement process is summarized as follows. The *Party* section consists of three parties, that is "Service Provider" and "Customer" as the signatory parties and "Monitoring Solution" as a supporting party. The sections concerning service definitions and obligations have been combined. The whole section of *ServiceDefinition* has been aggregated as a *ServiceLevelObjective*. The meaning of an SLO remains the same, just being represented in a less granular manner. What has been an *ActionGuarantee*, is now being shown as a *Penalty*. A penalty is actually a specific form of an action guarantee, namely being a monetary compensation from the SP to the customer when the associated *ServiceLevelObjective* is not met.

In contrast to most SLAs, `SLAMer` only allows monetary compensations rather than service credits. This is due to the fact that all payments happen automatically relying on the transfer of locked cryptocurrencies in the SC. Nevertheless, the design of `SLAMer` is able to accommodate service credits compensations by not considering the monetary compensation and registering, in the SC, the information that an SLA violation has occurred, the percentage of service credits that the customer is entitled to, and whether the service credits were claimed or not.

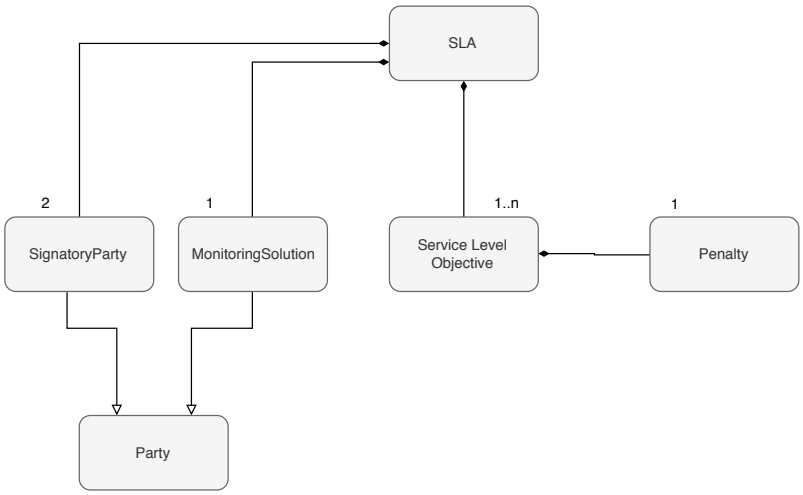

Figure 4.3: Simplified UML class diagram

4.3 SLAMer Data Flow

Before delving into implementation details, one must understand the high level interaction and the involved actors in SLAMer. Figure 4.4 depicts this interaction in a UML sequence diagram. The main actors are *ServiceProvider*, *Customer*, *MonitoringService* and *SmartContract*.

The main process runs as follows. The *ServiceProvider* creates the SLA by entering all the details as negotiated with the customer. During this process, the *MonitoringService* is registered by entering its wallet details. This wallet details are later needed to connect to the SC. To make sure that the SP does not enter parameters that do not correspond to the agreement, a review process is started. The *ServiceProvider* sends the SLA to the *Customer* who in turn checks if everything is correct. If this is not the case, he rejects the SLA with a note on what is wrong. The SP has to revise the SLA and send it again for review. This process is repeated until the *Customer* accepts it.

The *ServiceProvider* can now deploy the SC on the Ethereum blockchain. The *Customer* gets notified as soon as the *SmartContract* is deployed. He/She then deposits the agreed price in ether in the *SmartContract*. This amount is locked in the contract for the duration of its validity. This deposit activates the SC which is then ready to receive monitoring data.

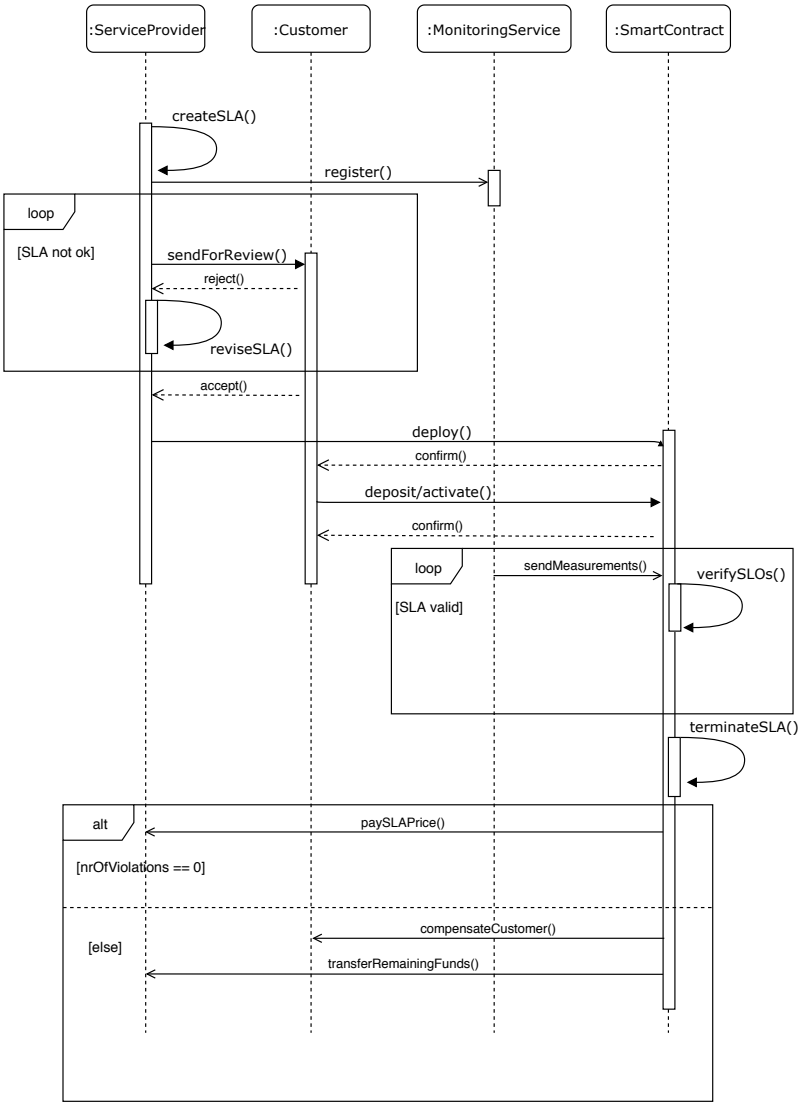

Figure 4.4: UML sequence diagram of the main data flow

The *MonitoringService* starts to send measurement data to the *SmartContract* by connecting to the Ethereum blockchain with the wallet details defined at the point of registration. As long as the SLA is valid, the *SmartContract* periodically verifies all the SLOs to make sure all the performance levels are met. There are two conditions that must be met cumulatively for the SLA to be valid, the first being that it is between the start and end date, the second being that the number of violations has not reached the violation threshold. The latter depends on the exact terms defined in the SLA. For example, the SLA can define the availability of a web server of 99.99% in one year, but if the actual up-time is less than 95%, the SLA is terminated, even if the year is not over yet. Any performance level between 95% and 99.99% will not terminate the SLA but will give a defined amount back to the customer at the end of the SLA lifetime.

Thus, when the SLA becomes invalid, the *SmartContract* will terminate the SLA. If the SLA has never been violated, the entire amount will be transferred to the *ServiceProvider*. Otherwise, a compensation value is calculated and deducted from the SLA price and sent

back to the *Customer*. The remaining funds will then be transfered to the *ServiceProvider*.

4.4 SLAMer SLA State Management

SLAMer introduces additional states complementing the SLA lifecycle defined in Chapter 2. These states are based on the SLA lifecycle states defined by IBM [28] and support the management of the lifecycle phases. The possible states in which an SLA can be and its relation to the lifecycle phases are presented in Table 4.1.

Lifecycle Phase	States	Description
Definition	Identified	The SP has entered the service requirements and the customer but has not yet submitted the SLA
Establishment	Requested	The SP finished registering the SLA and submitted it to the customer
	Accepted	The customer has accepted the terms submitted by the SP
	Rejected	The customer rejected the terms submitted by the SP
	Pending Deployment	The SC is being created and SLOs are being added; the confirmation from the blockchain is pending
	Deployed	The SC is deployed and ready to be activated by the customer
	Pending Deposit	The deposit is pending until the blockchain confirms this transaction
	Failed	Something went wrong with the SLA and cannot proceed to the next state
Monitoring	Active	The SLA is running and being monitored
Termination Penalty Enforcement	Inactive	The SLA has expired or has been terminated due to violations

Table 4.1: SLA Lifecycle and possible states

The reason why such states were introduced is to have a distinction inside the lifecycle phases itself. Notably, the *Establishment* phase requires a more granular view. When the SLA enters the *Establishment* phase, the review process starts between the customer and the SP (see Section 4.4). Before the SLA can be considered as active or ready for monitoring, the customer has to explicitly agree that the SLA is correct (*Accepted*). Furthermore, the SLA is activated in two steps, namely the SP deploying the SC on the blockchain and the customer depositing the funds on the contract. Both of these steps are direct interactions with the blockchain and require a confirmation before the process can proceed. Since this takes some time, transactions are pending while the confirmation is outstanding. If an error occurs during deployment or activation, the SLA is set to *Failed*. Without this status, the SLA would remain in its previous state not being able to proceed to the next state. Further, the involved parties would not be aware that the SLA has failed.

Concerning the two lifecycle phases *Termination* and *Penalty Enforcement*, they define two different endpoints of the SLA lifecycle, one stating that the SLA has ended normally *i.e.,* it expired after its end date, and the other indicating an abnormal abortion. Nevertheless, the phases both lead to the fact that the SLA is no longer in effect and thus, becomes *Inactive*.

These steps considered, a finer distinction is useful in order to make the transitions from one phase to the next more reliable and secure. The introduced steps serve as checkpoints to avoid invalid transitions *e.g.,* the customer rejects the SLA and the SP deploys it anyway. What's more, SLAMer can read the state of the SLA in order to perform access checks on which party can perform which actions, if any, in which state.

4.5 SLA SC

The SC was based on the definition provided by [9], where the SC implements functions to manage SLAs and compensate customers in a dynamic manner. However, in `SLAMer`, the compensation occurs when a violation is detected and the SLA finishes. Nevertheless, it is possible to implement the compensation value calculation based on each SLO in a dynamic manner.

Function	Parameters	Access Control
Create SC	customer, monitoringService, price, daysOfValidity	SP
`addSLO`	id, *	SP
`deposit`	value	customer
`verify`	measurement	Monitoring Service
`terminateSLA`	None	(Customer and SP) or Monitoring Service
`compensateCustomer`	None	Monitoring Service

Table 4.2: Defined SC Functions and Access Control

The logical flow occurs as follows. Firstly, the SP deploys the SC after both parties agreed on the SLA (see function *Create* in Table 4.2). Secondly, the customer deposits Ether in the height of the SLA price, which is then locked in the contract until the SLA expires or terminates. This deposit activates the SLA so that all parameters can be checked against incoming measurement data. Thirdly, when the SLA expires or terminates due to violations, the Ether locked in the contract is relieved to the SP deducted by any compensation. The compensation gets transferred back to the customer. In the following items, the functions in Table 4.2 are discussed.

- **Create**: Can only be triggered by the SP. This is ensured by a modifier in the SC. The parameters *customer* and *monitoringService* refer to their Ethereum wallet addresses.

- **addSLO**: After the SC is created, the SP adds each SLO separately. The function name *addSLO* is a placeholder name for all the SLOs. Each function is named after the pattern `add[sloType]`. Every function accepts the `id` parameter which serves as an unique identifier inside the SC. The asterisk (*) indicates that additional parameters are accepted, but vary from SLO to SLO, depending on its type.

- **deposit**: The ‘`deposit` function is callable for anyone, however, it is only executed if the sender’s address matches the customer’s address. Access control is also ensured by modifiers. The *value* parameter refers to the amount of Ether that is sent to the SC, which must be equal to the amount defined in the SLA.

- **verify**: The verification of the SLOs can only be accessed by the monitoring service. This is to prevent the parties to send wrong data on their behalf. For instance, the SP could send data that complies to the SLA in order to obfuscate that in reality the SLA is not met and prevent its termination. The same applies to the customer.

- **terminateSLA**: The termination of an SLA is triggered by the monitoring service, more precisely by the data it sends. The expiration also terminates the SLA and transfers the ether in the SC to the SP and to the customer in case of a violation. The *Customer* and *SP* are in parenthesis because in `SLAMer` both can trigger this function manually for testing and demonstration purposes. This behaviour can be restricted in the SC.

- **compensateCustomer**: This function is triggered by the `terminateSLA` function, but only if there are violations of the SLOs. It takes a specific amount of the deposited cryptocurrencies and transfers it to the customer's account.

4.6 Implementation

As discussed in Section 4.1, `SLAMer` consists of three main parts, namely the GUI, the backend and the SCs on the Ethereum blockchain. Details about each parts implementation are described in the following subsections.

4.6.1 `SLAMer` Backend

The backend hosts the main business logic in order to create and manage SLAs. It was written using the Java Spring framework [29], which is a framework that allows the quick development of state-of-the-art solutions. It allows to build applications according to different patterns. In the context of `SLAMer`, Spring helped in managing all the different business services and sharing single instances of them across the application using dependency injection. With this, a RESTful API was developed which listens to requests from different clients. Using Spring's security integration, all the REST endpoints can be secured against unauthorized or unauthenticated users and unknown origins.

All the data is stored in a PostgreSQL database [30]. For this, the database library jOOQ was used [31]. This enabled code generation out of the database schema, which resulted in being very helpful when changing tables or attributes in the database. Also, jOOQ, which stands for **Java Object Oriented Querying**, allows building SQL queries with java functions instead of executing plain SQL query strings. This helps to prevent syntax and type mapping errors due to database migrations. Moreover, jOOQ is easily integrated with Java Spring.

To connect to the Ethereum blockchain, the Java web3j library was utilized [32]. This library not only simplifies the process of deploying SCs to the blockchain but also to call all the SC functions by abstracting the creation of transactions and private key signing as well as the mapping from data types in Java and Solidity, *e.g.,* `BigInteger` to `uint` and vice versa. For a SC to be deployed, the Solidity source code first has to be compiled. This results in having two additional files, namely an Application Binary Interface (ABI) file and a binary file. With these two files, web3j generates a Java wrapper which makes the interaction possible and easy to use.

4.6.2 SLA SC

The SC was implemented using Solidity, which is a Turing-complete language provided by the Ethereum blockchain. Turing-completeness is required by `SLAMer` because there are functions *e.g.,* the calculation of the compensation, that require more complex functions.

Listing 4.1 shows the structure of an SLA. It consists of the signatory parties `serviceProvider` and `customer` which both represent the respective Ethereum addresses. This enables both parties to receive ether. The `monitoringService` acts as the supporting party. Furthermore, all the SLOs are stored as a mapping. Here the ID of each SLO serves as the key to access the corresponding object. The state attributes `paid`, `terminated` and `status` are used for checks to avoid unauthorized manipulations of the SLA. `status` can have one of three values, namely 3, 5 and 6. These codes stand for "Accepted", "Active" and "Inactive" respectively. `price` is the service price of the SLA. Since the SLA is only valid during a period of time, a small struct called `validity` stores the information about start and end time. The SLA instance in the SC is referenced by the name `deployedSLA`.

```
1  struct SLA {
2      address payable serviceProvider;
3      address payable customer;
4      address monitoringService;
5      mapping(uint => Slo) slos;
6      uint price; // price for the service
7      bool paid; // is it paid?  = false
8      bool terminated;
9      uint status; // 3 (Accepted), 5 (Active), 6 (Inactive)
10     validityPeriod validity;
11 }
12
13 SLA deployedSLA;
```

Listing 4.1: SLA Struct

The implementation logic of SLOs consists of several contracts. By considering a contract as equivalent to a class, the structure can be represented as an UML class diagram as shown in Figure 4.5. Every SLO can be verified; thus, it implements the *Verifiable* interface. Since `SLAMer` currently only supports three types of SLOs, the SC can also only support these three types. *Uptime*, *Throughput* and *AverageResponseTime* implement the specific logic of verifying each SLO type.

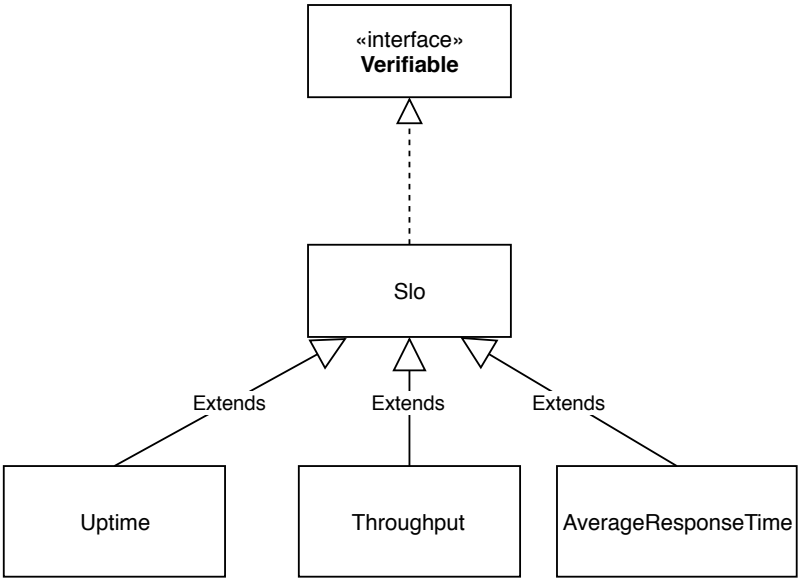

Figure 4.5: UML class diagram of SLOs

Listing 4.2 shows the SC code of the interface and SLO. The `Slo` contract only extends the interface by the fields `id`, `violations` and `maxViolations`. The `Slo` is an abstract representation stating that any SLO can be violated a certain amount of times and therefore provides the `Violated` event. Since an SLO is added to an SLA, there is also an `SloAdded` event to confirm that the SLO has been added successfully.

```
1  interface Verifiable {
2      function verify(uint _measured) external returns (bool);
3  }
4
5  contract Slo is Verifiable{
6          uint id;
7          uint violations;
8
9          uint maxViolations = 5;
10
11         event SloAdded();
12         event Violated();
13
14         constructor(uint _id) public {
15             id = _id;
16             violations = 0;
17         }
18 }
```

Listing 4.2: SLO contract and the *Verifiable* interface

The SLO contract does not yet implement the `verify` function of the `Verifiable` interface. Every type of SLO is a separate contract that extends an abstract `Slo`. The concrete SLO types implement this function as it is illustrated in Listing 4.3. It shows the types `Uptime`, `AvrgResponseTime` and `Throughput` on lines 1, 14 and 28 respectively. They all extend the `Slo` by their characteristic attributes which will be verified during

the monitoring phase. Listing 4.3 only shows the stubs of the `verify` functions, as they are explained in detail further in this chapter.

```
contract Uptime is Slo {
    uint percentageOfAvailability;

    constructor(uint _id, uint _availability) Slo(_id) public {
        percentageOfAvailability = _availability;
        emit SloAdded();
    }

    function verify(uint _measured) public returns (bool) {
        // verification logic
    }
}

contract AvrgResponseTime is Slo {
        uint avrgResponseTimeValue;

        constructor(uint _id, uint _responseTime)
        Slo(_id) public {
            avrgResponseTimeValue = _responseTime;
            emit SloAdded();
        }

        function verify(uint _measured) public returns (bool) {
            // verification logic
        }
}

contract Throughput is Slo {
        uint dataSize;
        uint thresholdValue;

        constructor(uint _id, uint _dataSize, uint _thresholdValue)
           Slo(_id) public {
            dataSize = _dataSize;
            thresholdValue = _thresholdValue;
        }

        function verify(uint _measured) public returns (bool) {
            // verification logic
        }
}
```

Listing 4.3: The concrete SLO contract implementations

The main functions presented in Table 4.2 are described as follows. “Create” is actually the `constructor` function but for the sake of understanding this name was chosen. When a SP deploys the SC, the `constructor` function (see Listing 4.4) is called. `SLAMer` gets the wallet address from the customer and the monitoring service and passes this information together with the number of days the SLA is valid and its price to the constructor. Before anything happens, it is assured that no party goes under the same address as an other party. In other words, the customer or monitoring solution can never create the SC because `SLAMer` will always send the customer and monitoring service address as the

parameters. The address of the SP, who is the sender, is stored as the `serviceProvider`. Once the initialization has completed, the `ContractCreated` event is emitted to notify the SP that the SC has been created successfully.

```
1     constructor(address payable _customer, address
          _monitoringService,uint _price, uint _daysOfValidity) public{
2         require(msg.sender != _customer, "The SP must not be the
              customer");
3         require(msg.sender != _monitoringService, "The SP must not
              be the monitoring service");
4         deployedSLA.serviceProvider = msg.sender;
5         deployedSLA.customer = _customer;
6         deployedSLA.monitoringService = _monitoringService;
7         deployedSLA.price = _price;
8         deployedSLA.paid = false;
9         deployedSLA.terminated = false;
10        deployedSLA.validity.daysOfValidity = _daysOfValidity;
11        emit ContractCreated();
12    }
```

Listing 4.4: constructor, named "Create"

In the current state, there are no SLOs in the `deployedSLA` yet. In the process of creation, `SLAMer` has to add each SLO individually but has to wait until `ContractCreated` confirms that the contract was created. `SLAMer` evaluates each type of SLO in the SLA and calls the corresponding functions. Listing 4.5 shows the three functions that the SP calls when adding SLOs. Since an SLA can have any number of SLOs, `SLAMer` notifies the SC that all SLOs have been added, by calling the `confirmComplete` function (see Listing 4.5, line 10). The SLA gets initialized with a `status` of 3. In `SLAMer`, 3 stands for "Accepted" which describes the state where both parties have agreed on the SLA but it is not yet active.

```
1     function addUptime(uint _id, uint _availability) public onlySP{
2         deployedSLA.slos[_id] = new Uptime(_id, _availability);
3     }
4
5     function addAvrgResponseTime(uint _id, uint _responseTime)
          public onlySP {
6         deployedSLA.slos[_id] =
7                         new AvrgResponseTime(_id, _responseTime);
8     }
9
10    function confirmComplete() public onlySP {
11        deployedSLA.status = 3; // Accepted
12        emit ContractComplete();
13    }
```

Listing 4.5: SLO addition functions

After the SP successfully deployed the SC, it is the customer's task to deposit Ether in the contract to activate it. The `deposit` function can only be executed by the customer, as it is ensured by the modifier in Listing A.1 at line 2. Several checks are made, namely that the contract is not already active, and the amount of ether sent by the customer must correspond exactly to the `price`. Further, the status of SC must be 3 (Accepted). The

start and end time of the SLA is set by taking the current timestamp as the start time and adding the amount of days to it to determine the end time. This function can only run once, since it would not pass the `require` statements a second time. After running successfully, `SLAMer` gets notified by the `CustomerDeposit` event containing the customer address and the amount of ether he has sent. The SLA is now "Active" (see listing A.1, line 12) and ready to validate the SLOs.

```
1          function deposit() public payable
2                      onlyCustomer returns (bool) {
3          require(!isActive(), "The contract must not be active yet");
4          require(deployedSLA.price == msg.value,
5                                  "The value must equal the SLA price");
6          require(deployedSLA.status == 3,
7                                  "The SLA must have status 3 (Accepted)")
                                      ;
8          deployedSLA.paid = true;
9          deployedSLA.validity.startTime = now;
10         deployedSLA.validity.endTime = now + (deployedSLA.validity.
               daysOfValidity * secondsPerDay);
11         emit CustomerDeposit(msg.sender, msg.value);
12         deployedSLA.status = 5; // Active
13         return true;
14     }
```

Listing 4.6: `deposit` function

Once the SLA is "Active", the monitoring service can start measuring the SLOs defined in the SLA. Those values are sent to the *SLAMerAPI* (see Listing 4.1 for reference). `SLAMer` retrieves the correct SC address and calls the correct functions depending on the SLO type (see Listing 4.7). The verification logic is illustrated using the *Average Response Time* SLO. `SLAMer` sends the measured response time together with the ID of the SLO to the function. First, the function can only be executed by the monitoring solution. This is ensured by the `onlyMonitoringService` modifier. Second, it is checked if the SLA is still valid, both in terms of validity and number of violations. After retrieving the corresponding SLO, its `verify` function is called (see Listing 4.7). Should the `verify` function return, hence verification fails, the SLA is terminated (this process is explained in detail further on the thesis).

```
1      function verifyAverageResponseTime(uint _sloId, uint _measured)
           public onlyMonitoringService {
2          checkValidity();
3          Slo avrgResTime = deployedSLA.slos[_sloId];
4          if (!avrgResTime.verify(_measured)) {
5              terminateSLA();
6          }
7      }
```

Listing 4.7: Verification functions

As mentioned above, the `verify` function checks the measured response time value against the value specified in the SLA. As it is shown in Listing 4.8, a counter adds the number of violations. The maximum number of tolerated violations is currently set to five. When exceeding this threshold, the `Violated` event is emitted to `SLAMer` . The value of

`maxViolations` is just an arbitrary number. Of course it could be made more dynamic, for instance specify a percentage as a threshold (*e.g.,* 90% of the measured response times should be less than 500 ms). For testing purposes and the sake of simplicity, a static value was chosen.

```
1
2       uint maxViolations = 5;
3
4       function verify(uint _measured) public returns (bool) {
5               if (_measured > avrgResponseTimeValue) {
6                   violations += 1;
7                   if (violations >= maxViolations) {
8                       emit Violated();
9                       return false;
10                  }
11              }
12              return true;
13          }
```

Listing 4.8: Average Response Time *verify* function

4.6.3 Graphical User Interface

The graphical user interface was developed using the Angular framework [33] created and maintained by Google. Angular is a JavaScript framework which uses the Model-View-Controller (MVC) pattern and organizes the frontend of an application into reusable components. Angular provides its own language, called Typescript, which is a super set of JavaScript that contains more functionality than JavaScript itself. Furthermore, it contains useful modules for routing and HTTP. Especially the HTTP module is useful for easily constructing asynchronous requests and send them to the backend API. In terms of resource management, angular provides so-called lifecycle methods for each component. These create, place and remove these components from the Document Object Model (DOM). Coupled with that, it allows to perform specific operations during specific lifecycle steps, for instance fetch information from an API before the component is displayed to the user.

Functionality

The frontend is composed of three parts. *(i)* The *Home* tab which provides an overview over a users SLAs, *(ii)* the *Create SLA* where a user can create a new SLA in the role of the SP – This starts a process where all the SLOs are registered as well as the monitoring solution – and *(iii)* the *Notifications* tab informs the user about status changes and actions that the user has to perform.

Figure 4.6 shows the screen of the overview that the users face when they log into `SLAMer`. It contains a table presenting general information about all the SLAs a party has, either as a SP or a customer. Every SLA is given a title to better identify an SLA. On the top, a pie chart shows how many SLAs are in which lifecycle phase (see Section 2.1.1 for more

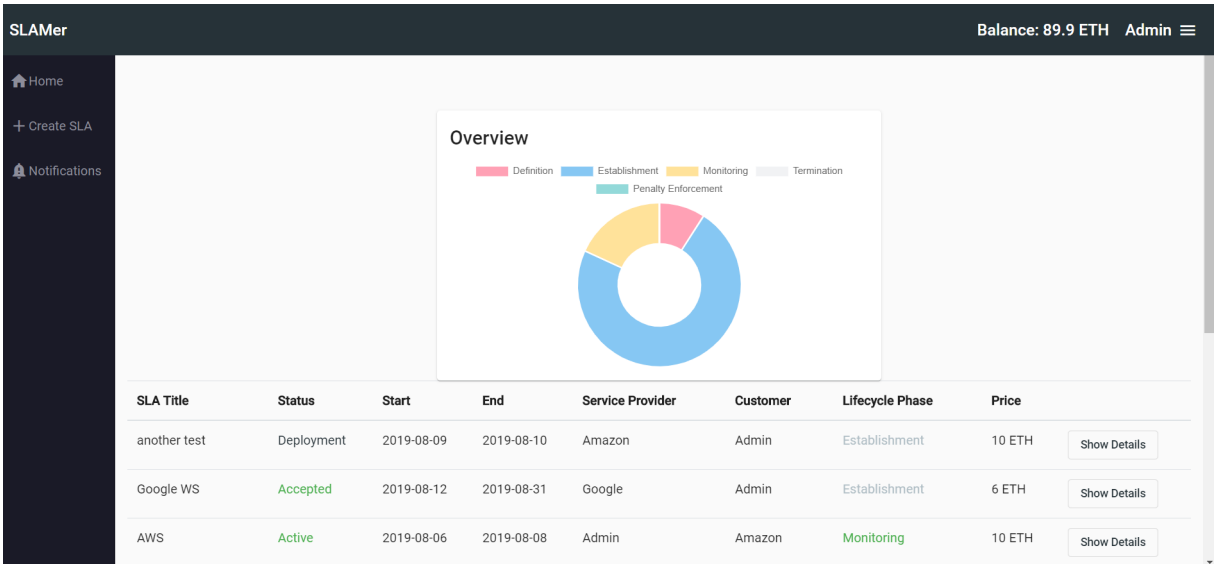

Figure 4.6: Home tab with the SLA overview

details). From here, the details of each SLA can be seen by clicking on the respective button on the right hand side of each row. Furthermore, on the top right of the page, the current balance is showed if the wallet details are registered.

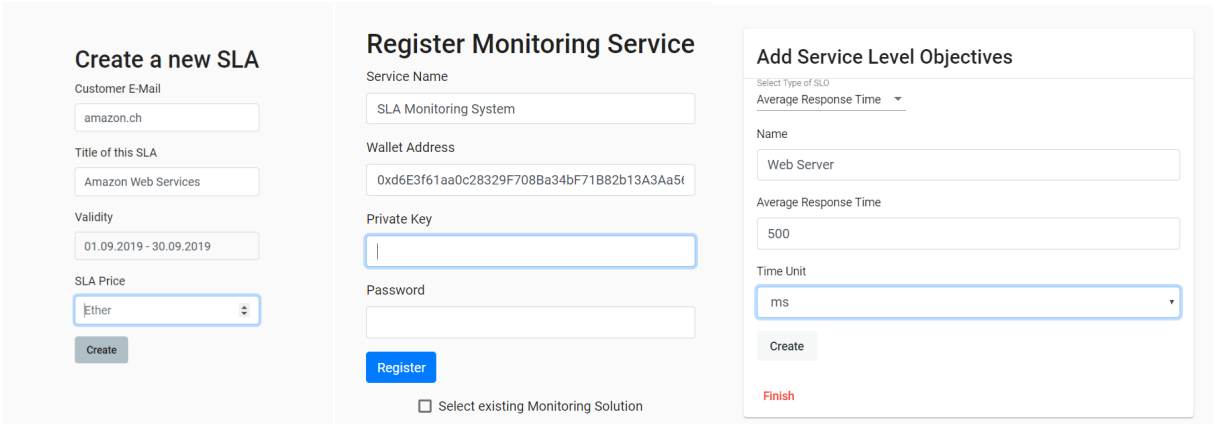

Figure 4.7: The first three steps of creating an SLA

The second tab, *Create SLA*, lets a SP start the SLA creation process. This process is depicted in Figure 4.7 and Figure 4.8 and consists of four steps. It starts with specifying the customers email address and the title of the SLA. The start and end time of the SLA are chosen as well as the total service price.

In step two, the SP has to register a new monitoring service which will be responsible for this SLA. This is done by entering the wallet address and private key of the service. This is due to the fact that wallet credentials are needed in order to interact with the SC. However, the SP can also choose an already existing monitoring service by clicking on the respective checkbox at the bottom.

The third step provides the possibility to add all the SLOs. The SP can choose between the types *AverageResponseTime*, *Uptime* and *Throughput* and fill in the corresponding

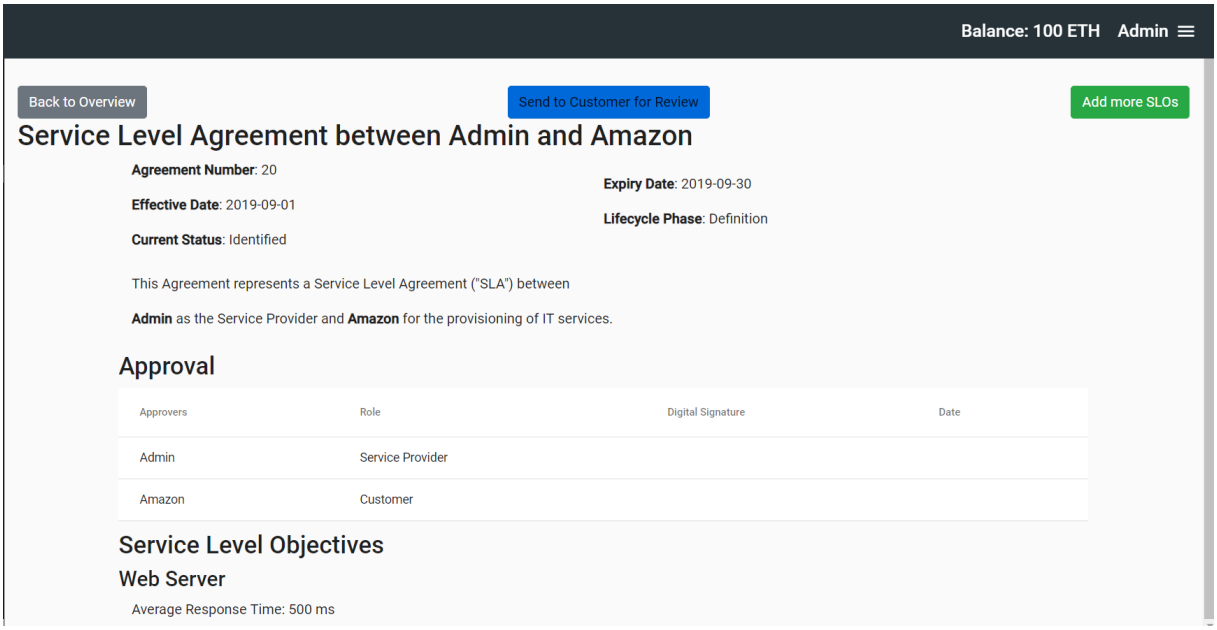

Figure 4.8: Step 4: Detailed overview over the created SLA

details. Once he finished adding the SLOs, he can go to the next workflow step by clicking *finish*.

In the last step (see Figure 4.8), the SP is provided a detailed overview over the current SLA. He can check if all the SLOs are included and click *add more SLOs* on the top right if something is missing. This will lead the SP back to step three. When the SLA is complete, he can send it to the customer for review by clicking on the respective button. This ends the process for the SP.

The last tab, *Notifications*, is shown in Figure 4.9. Both parties get notified about actions required to perform. As a customer, he has to *Review* a new SLA and either accept it or reject it with a note on what is wrong. The SP on the other hand has to *Revise* an invalid SLA, which moves the SLA back to the customer. When the SLA is accepted by the customer, the SP can *Deploy* the SLA. This is the moment where the SC gets created on the Ethereum blockchain and populated with all the data from the SLA. This leads to the last action to be performed by the customer, namely *Activate*. This is where he deposits Ether in height of the SLA price on the SC and makes it ready to be monitored.

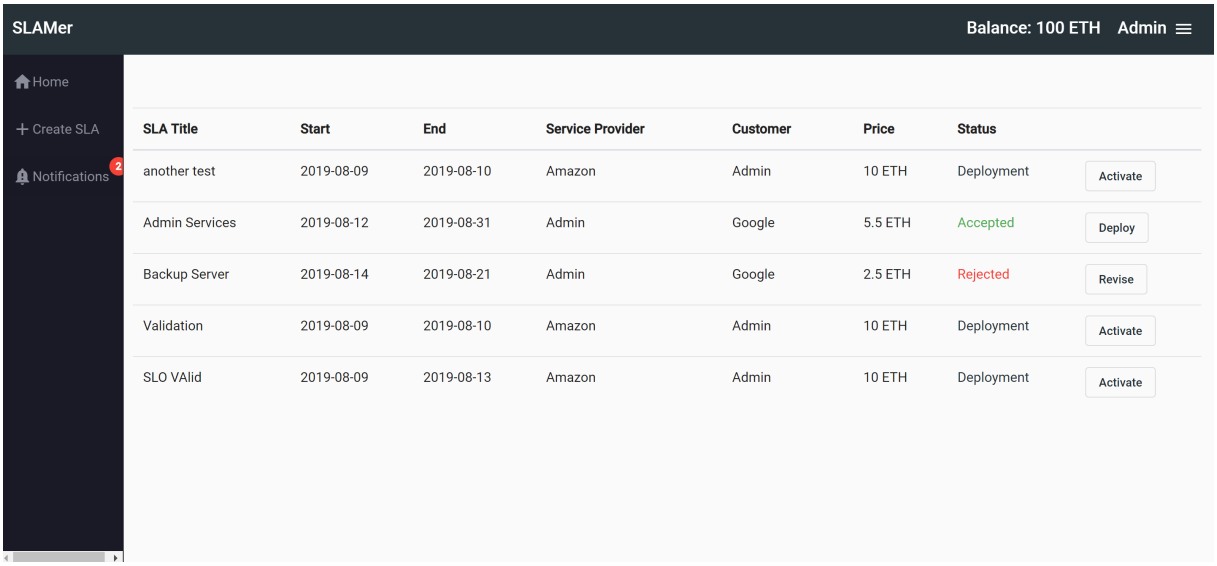

Figure 4.9: Notifications tab with required actions

Chapter 5

Evaluation and Discussion

`SLAMer` was evaluated regarding three aspects. Section 5.1 is dedicated to the economic aspect in terms of costs. The management aspect is evaluated and discussed in Section 5.2, taking into account security-related elements as well. In Section 5.3, usability and performance of `SLAMer` are discussed with regard to the blockchain performance.

For this purpose, Ganache was used to simulate a private Ethereum blockchain. This tool runs independently from the web3 implementation. Ganache is a standalone application that comes with 10 pre-configured accounts per default, each having an initial balance of 100 ETH. What's more, one can enable auto-mining which means that transactions are processed instantaneously. It can also be turned off and one can define a custom mining block time to simulate real-world behaviour.

5.1 Economical Evaluation and Discussion

Since the deployment and interactions with the SC that alters its state require a payment of a certain gas fee, this section analyzes these costs. Table 5.1 presents an overview over the SC functions with the gas consumption and its equivalent value in USD. The values for the consumed gas are mean values taken from 5 transactions. In order to retrieve these values, a SC was deployed from `SLAMer` to the Ganache blockchain. All the functions listed in Table 5.1 were then triggered either from `SLAMer` or manually from the Remix IDE. Ganache has a transaction and block history, providing information such as the amount of gas consumed per transaction and per block. The transactions were executed at a fixed gas price of 5 Gwei (*i.e.,* 5 billion Wei) and an ether price of 191.07$ as of August 25 2019.

Function	Gas Consumed	Price [USD]
Create SC	1,154,153	$1.09645
`addSLO`	42,540	$0.04041
`deposit`	89,374	$0.08491
`verify`	21,656	$0.02058
`terminateSLA`	50,106	$0.0476
`compensateCustomer`	30,076	$0.02858

Table 5.1: Gas and Transaction Price Estimation

It is noticeable that the creation of the SC is the most expensive function. This emerges from the way transaction costs are composed in Ethereum, namely a fixed and a variable part. The EVM demands a fixed cost of 32,000 gas for a contract creation in addition to the 21,000 gas that every transaction costs. The remainder is the variable part which depends on the size of the contract code. Each byte of code consumes 200 gas *i.e.,* the more code a contract has the more expensive its creation is. All the other functions in Table 5.1 are normal transactions. These are composed of the fixed gas fee of 21,000 plus a certain fee for each operation [11].

The monitoring service needs to pay the gas fees every time it sends data to the SC. This results in a trade-off between cost and accuracy. The more often the service sends measurements, the more expensive it becomes. In the same time, the data is more accurate since the frequency of measurements leads to a lower chance of missing a violation. When increasing the time between the intervals, costs decrease but chances are higher that a violation is not detected. Clearly, this only applies to SLOs which need constant monitoring *e.g.,* availability. Other type of SLOs do not have this disadvantage as they only require verification on a specific occurrence *e.g.,* response time to a phone call only needs to be verified when a phone call is made.

To go further into detail, Equation 5.1 presents the calculation of the gas cost. Every verification in the SC requires a certain amount of gas ($Gas_{consumed}$), which is multiplied with the gas price. Additionally, this cost is multiplied by the number of SLOs that are verified. Equation 5.1 shows the calculation of the accumulated cost over time. The $SLA_{validity}$ represents the validity period of an SLA, which is divided by the $monitor_{granularity}$. *Monitoring Granularity* is defined by the period of time that a monitoring solution will inform the values to the SC. This results in the total number of verifications throughout the SLA lifetime. Multiplying this ratio by the Gas_{cost} results in the accumulated costs.

$$Gas_{cost} = Gas_{consumed} \times Gas_{price} \times SLO_{quantity} \tag{5.1}$$

$$Accumulated_{price} = \frac{SLA_{validity}}{monitor_{granularity}} \times Gas_{cost} \tag{5.2}$$

When plotting this equation with some sample data, one gets the chart depicted in Figure 5.1. The lines follow a linear equation, which is defined by $y = ax + b$, where a is the gradient and b represents where the curve intercepts the y-axis. For one SLO, three different levels of monitoring granularity were chosen (*e.g.,* 0.5 s, 1 s, and 2 s) for an SLA validity of 30 min. When taking the gas consumption of the verification function of 21,656 (see Table 5.1) and the current standard gas price as of August 2019 [34] of 5 Gwei, the gas cost of one verification is 0.00010828 ETH.

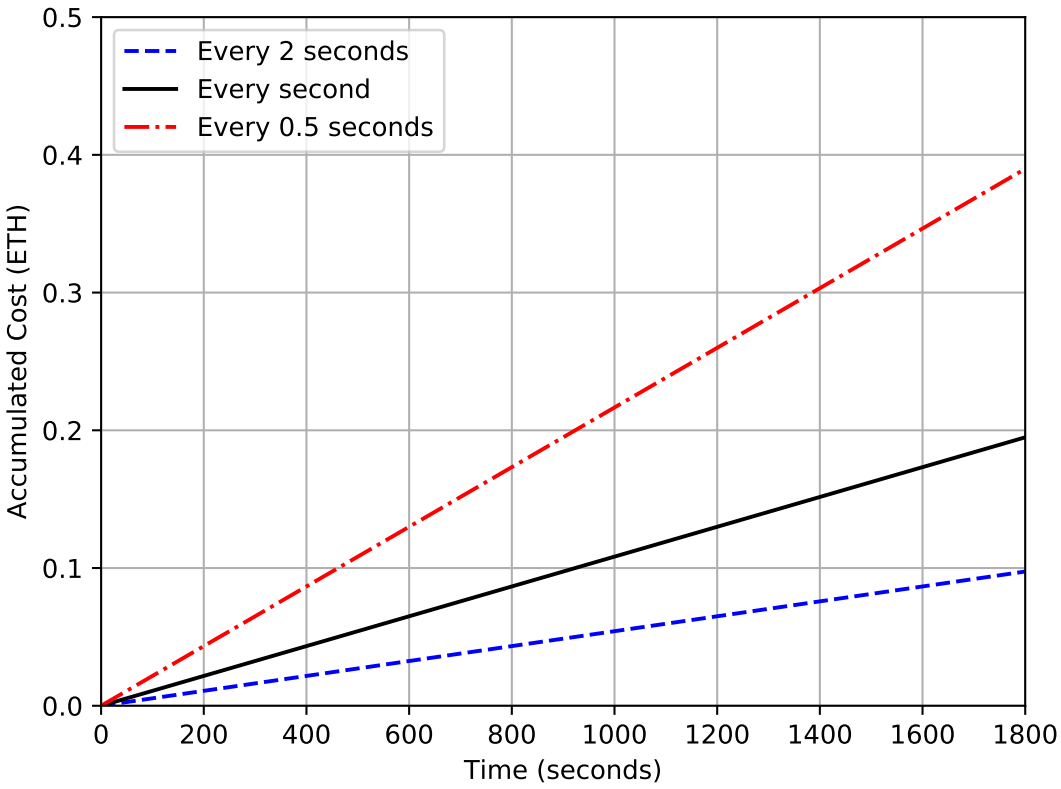

Figure 5.1: Accumulated Cost for Smart Contract verification over time

The level of the monitoring granularity has a direct impact on the slope a and acts as a lever. As can be seen in Figure 5.1, when monitoring the service in intervals of 2 seconds the accumulated costs amount to 0.1 ether. When measuring every second and every half second, the costs result to approximately 0.2 and 0.4 ether respectively. As of August 2019, verifying the SLO for 30 minutes every half second will cost about 74 USD.

For the plot in Figure 5.1 a fixed gas price is assumed. When monitoring for longer periods, the curve will not remain linear, since the gas price is under constant change as well as the ether price. Nonetheless, it should be quite obvious that having a monitoring granularity in the range of seconds is not feasible considering that SLAs usually have a validity of several months to a year. Keeping the same monitoring granularity would result in being quite expensive for the monitoring service. Thus, in order to reduce these costs, the monitoring granularity has to be decreased. As mentioned before, increasing the time between verifications will lead to a higher probability that violations are not detected. SPs and customers would need to find the optimum between costs and accuracy, so that

they can set the monitoring service to pay a reasonable price for an adequate monitoring granularity.

5.2 Management Discussion

One advantage of `SLAMer` is the data integrity provided by the blockchain. All the data stored in the SC, including the SC source code, cannot be altered once it is deployed. Likewise, this data is always available as long as one has the address of the SC, since it is not dependent on a single node of the blockchain. This means, SLAs and its data can be viewed without having to rely on `SLAMer`.

Concerning SLA management, penalties can be enforced without the customer having to manually prove a violation. This fact is helpful for both parties. Customers do not have to invest time in the investigation and data gathering in case of a violation, not to mention the avoidance of going through the service request process and waiting for a response, as well as potentially having to repeat this process should the request be rejected. The SP can benefit from this as well. In order to process all service requests, resources have to be allocated which manually work through these requests. This is a time-consuming endeavour because the data provided by the consumer to underpin his claim needs to be examined and compared to the SPs measurements. Those resources of the SP could be saved or used elsewhere.

One drawback of this solution is the centralization that occurs by relying on the database in `SLAMer`. Before the SC is deployed, the availability of the SLA data relies completely on the availability of the database itself. After the contract is deployed on the blockchain, both parties are able to retrieve the SC address. With this address parties can also interact with the SC without `SLAMer` because they can use another client for this. Naturally, they need to connect to the blockchain with their correct wallet for this.

Regarding the use of an external monitoring solution, two problems arise. First, the SP and customer need to be sure that the monitoring service is measuring the SLOs correctly and constantly. If the service sends wrong measurements, the SC could potentially terminate the SLA mistakenly or, in the opposite case, not detect a violation although the SLO is not met. Second, the monitoring service has to function properly during the entire lifetime of the SLA. In the case where the monitoring service is down, no measurements can be sent to the SC and thus, the SLA remains unverifiable. Furthermore, in the meantime the customer would need to monitor the SLA manually again, as well as prove any violations. Taking these two problems into account, the monitoring service remains as a TTP in the system.

Security-wise, there currently is an issue regarding the wallet credentials of users. Since the interaction with the Ethereum blockchain happens server side (backend), this implies that the server would need to pay for all the gas fees for all users in the system. To avoid this, parties have to provide their wallet address and private key, so that `SLAMer` can perform transactions on their behalf. This can result in a serious issue since the private key is intended to remain with the wallet owner only. Further, this adds an additional

layer of trust, because users have to be sure that `SLAMer` does not lose their private keys to an attacker.

The other possibility would be to shift the deployment and interaction of SCs to the client. In this case, both parties can pay their own fees and retain their private keys. Additionally, the aforementioned trust issue would not exist. Since all the important data will be immutably persisted in the SC on the blockchain, any data leak would not expose the users to any risk whatsoever. In order to interact with the SC from the client-side, users would need to install a browser add-on like MetaMask [35]. MetaMask allows users to have a wallet in the browser that facilitates interaction directly with the blockchain. This topic is mainly a design decision and should be possible to adapt if required.

5.3 Usability and Performance Discussion

In `SLAMer`, usability and performance are directly related to the performance of the underlying blockchain. Since every transaction to the SC needs to be mined first in order to be appended to the blockchain, this leads to a blockchain-dependent latency. Each interaction with the blockchain that requires a confirmation or modifies a state in the SC requires a certain time until the user knows whether it was successful or not. In the Ethereum blockchain the average block time is approximately 15 seconds [36]. This impacts all the interactions but mainly influences the process of deploying the SLA SC. In order for `SLAMer` to include all SLOs in the SC, it first has to wait for the confirmation that it has been created. After that, `SLAMer` can sequentially add SLOs and finalize by confirming that all SLOs have been committed (see Section 4.6.2).

When deploying the `SLAMer` SC in a test environment, as it was done in this thesis using Ganache, the customer needs to wait for at least 2 blocks until he/she is able to deposit the funds. Since Ganache allows to adjust the block time, this wait was not an issue in the evaluation. However, when working in a production environment (*i.e.,* deploying the SC in the Ethereum mainnet), one must wait several confirmations of the block containing the transaction until it can be considered as secure. According to the Ethereum white paper, 7 confirmations should be enough to consider a transaction as confirmed [37], which is about 2 minutes. Depending on the level of security, one can also require more confirmations in order to be sure that a transaction is not reverted. For example, the Coinbase trading platform requires 35 confirmations until a transaction is completed [38] *i.e.,* about 8 to 9 minutes. As mentioned in Section 2.3.1, higher gas prices have an impact on how fast a miner includes a transaction in a block; thus, the number of confirmations, the chosen gas prices as well as the network congestion determine the actual time until a transaction is completed.

To conclude, parties creating SCs in `SLAMer` do not have an immediate confirmation about its creation and therefore have to wait several minutes. As long as the creation of an SLA in a SC is not time-critical, this should not result in an issue.

Chapter 6

Conclusion and Future Work

The goal of this thesis was to investigate the employment of blockchain and SCs in the field of SLAs in order to design and implement a blockchain-based SLA management system. SLA is a topic widely researched and many authors have proposed solutions for its issues. However, blockchain-based SC is a novel concept; thus, being slowly applied in to solve trust issues and payment enforcements in the SLA context. SCs have been proposed for solving trust issues and payment enforcement. Starting from this point, `SLAMer` was designed, based on the WSLA language, and a PoC implemented. It covers all the lifecycle phases starting from the second, providing a GUI for both SP and customer to register, activate and monitor their SLAs.

The PoC of `SLAMer` was evaluated in terms of economic, management aspects as well as usability and performance. In terms of economic aspects, the costs of managing the SC (*e.g.,* deploy SC, include SLOs, and verify violations) resulted in being considerably high. Verifying measurements coming from a monitoring solution over the SLA lifetime requires continuous calls to the SC in specified intervals. There is a trade-off between cost and the possibility of missing a violation. Depending on the parties and SLAs, the verification intervals have to be set according to their needs.

Regarding management, there are benefits to both sides (SP and customer), such as the data integrity and enforcement of penalties. Drawbacks arise from the centralization by partially relying on the database in `SLAMer` and being dependent on the correctness of the data provided by the monitoring solution. Further, parties have to provide both public and private keys to `SLAMer` so that `SLAMer` can sign and send transactions on their behalf, since the interaction with the blockchain is performed server-side. However, this interaction can be shifted to the client which allow parties to manage their private keys. By doing this, parties also rely less on the availability and security of `SLAMer` due to the public accessibility of their contracts on the blockchain.

The employment of a public blockchain introduces challenges in terms of performance and usability. Because `SLAMer` interacts closely with the Ethereum blockchain, its performance is directly related to the block time of Ethereum. Thus, users cannot expect immediate confirmations of deployed SCs since they have to wait until the blocks containing their transactions are confirmed by the network which can take several minutes, which hinders the usability of `SLAMer`.

In conclusion, `SLAMer` can help in enforcing payments according to agreed terms in an SLA. The cost of using SCs are is still expensive compared to a central database and the main challenge regarding the employment of SC in this area. Further research is needed to find a more cost-effective way of verifying SLOs. A possible option is finding a blockchain that supports Turing-complete SCs and demands lower transaction fees. Even though there are challenges, the approach presented in this thesis is feasible and, based on the evaluation performed, the blockchain employment to address issues related to SLA compensation is beneficial.

Since `SLAMer` is a PoC, it can be improved in the future. Future work is planned to consider *(i)* multi-party SLAs (*i.e.,* multiple SPs or customers), *(ii)* implement the complete WSLA framework, with SLA definitions and parties, *(iii)* include the support of monitoring a great number of different SLOs, and *(iv)* integration with different Operational and Business Support Systems (BSS). The SCs can be improved in the future as well. A classification of different SLOs could help in implementing different generic contracts instead of creating a separate contract for each SLO type. This would make the SCs more dynamic and potentially simplify the calculation.

Bibliography

[1] Salman Taherizadeh et al. "Monitoring Self-Adaptive Applications within Edge Computing Frameworks: A State-of-the-Art Review". *Journal of Systems and Software* 136 (2018), pp. 19–38. ISSN: 0164-1212. URL: http://www.sciencedirect.com/science/article/pii/S016412121730256X.

[2] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. [On-line], http://bitcoin.org/bitcoin.pdf last visit May 20, 2019. 2009.

[3] Daniel Macrinici, Cristian Cartofeanu, and Shang Gao. "Smart Contract Applications Within Blockchain Technology: A Systematic Mapping Study". *Telematics and Informatics*. Elsevier, 2018.

[4] A. S and C. K. "Monitoring and Management of Service Level Agreements in Cloud Computing". *IEEE International Conference on Cloud and Autonomic Computing (ICCAC 2015)*. Sept. 2015, pp. 204–207.

[5] Azure. *SLA for Virtual Machines*. [On-line], https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_8/ last visit July 30, 2019.

[6] Amazon Web Services. *Amazon Compute Service Level Agreement*. [On-line], https://aws.amazon.com/compute/sla/ last visit July 30, 2019.

[7] Azure. *Google Compute Engine Service Level Agreements*. [On-line], https://cloud.google.com/compute/sla last visit August 2, 2019.

[8] A. Maarouf, A. Marzouk, and A. Haqiq. "Practical modeling of the SLA life cycle in Cloud Computing". *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*. Dec. 2015, pp. 52–58. DOI: 10.1109/ISDA.2015.7489170.

[9] E. J. Scheid et al. "Enabling Dynamic SLA Compensation Using Blockchain-based Smart Contracts". *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019)*. Apr. 2019, pp. 53–61.

[10] Sarah Underwood. "Blockchain Beyond Bitcoin". *Commun. ACM* 59.11 (Oct. 2016), pp. 15–17. ISSN: 0001-0782.

[11] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger" ().

[12] Nick Szabo. "Formalizing and Securing Relationships on Public Networks". *First Monday* 2.9 (1997). URL: https://ojphi.org/ojs/index.php/fm/article/view/548.

[13] Cardano Foundation. *Cardano*. [On-line], https://www.cardano.org/en/home/ last visit August 25, 2019.

[14] Ethereum. *Ethereum Remix*. [On-line], https://remix.ethereum.org last visit August 21, 2019.

[15] Ethereum. *Serpent*. [On-line], https://github.com/ethereum/serpent last visit August 18, 2019.
[16] Ethereum. *Vyper*. [On-line], https://github.com/ethereum/vyper last visit August 18, 2019.
[17] Ethereum. *LLL PoC 6*. [On-line], https://github.com/ethereum/aleth/wiki/LLL-PoC-6/d64849ce last visit August 18, 2019.
[18] Ethereum. *Solidity*. [On-line], https://solidity.readthedocs.io/en/v0.5.11/ last visit August 18, 2019.
[19] Heiko Ludwig et al. *Web Service Level Agreement (WSLA) Language Specification*. Jan. 2003.
[20] A. Dan et al. "Web services on demand: WSLA-driven automated management". *IBM Systems Journal* 43.1 (2004), pp. 136–158. ISSN: 0018-8670.
[21] S. Zhang and M. Song. "An Architecture Design of Life Cycle Based SLA Management". Vol. 2. Feb. 2010, pp. 1351–1355.
[22] F. Seyed Mostafaei, N. Amani, and P. Hajipour. "Proposing a new QoS/SLA Management Model by Regulatory Authority". *International Symposium on Telecommunications (IST 2010)*. Tehran, Iran, Dec. 2010, pp. 508–512.
[23] Eder John Scheid and Burkhard Stiller. "Leveraging Smart Contracts for Automatic SLA Compensation - The Case of NFV Environment". *IFIP 12th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2018)*. Munich, Germany, June 2018, pp. 70–74.
[24] H. Nakashima and M. Aoyama. "An Automation Method of SLA Contract of Web APIs and Its Platform Based on Blockchain Concept". *IEEE International Conference on Cognitive Computing (ICCC 2017)*. Honolulu, HI, USA, June 2017, pp. 32–39.
[25] H. Zhou, C. de Laat, and Z. Zhao. "Trustworthy Cloud Service Level Agreement Enforcement with Blockchain Based Smart Contract". *IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2018)*. Dec. 2018, pp. 255–260.
[26] E. Di Pascale et al. "Smart Contract SLAs for Dense Small-Cell-as-a-Service". *CoRR* abs/1703.04502 (2017). Available at http://arxiv.org/abs/1703.04502 Accessed 29 March, 2019. arXiv: 1703.04502.
[27] R. B. Uriarte, R. de Nicola, and K. Kritikos. "Towards Distributed SLA Management with Smart Contracts and Blockchain". *IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2018)*. Dec. 2018, pp. 266–271.
[28] IBM. *Service level agreement lifecycle*. [On-line], https://www.ibm.com/support/knowledgecenter/en/SSWLGF_8.0.0/com.ibm.sr.doc/rwsr_gep_sla_life_cycle.html last visit August 16, 2019.
[29] Spring. *Spring Framework*. [On-line], https://spring.io/ last visit August 3, 2019.
[30] PostgreSQL. *PostgreSQL*. [On-line], https://www.postgresql.org/ last visit August 3, 2019.
[31] Lukas Eder. *jOOQ*. [On-line], https://www.jooq.org/ last visit August 3, 2019.
[32] Web3 Labs. *Web3j*. [On-line], https://web3j.io/ last visit August 3, 2019.
[33] Google Inc. *Angular*. [On-line], https://angular.io/ last visit August 11, 2019.
[34] ETH Gas Station. *Gas-Time-Price Estimator*. [On-line], https://ethgasstation.info/ last visit August 16, 2019. 2019.

[35] MetaMask. *MetaMask*. [On-line], https://metamask.io/ last visit August 15, 2019.

[36] Ethereum. *Etherscan*. [On-line], https://etherscan.io/chart/blocktime last visit August 26, 2019.

[37] Ethereum. *A Next-Generation Smart Contract and Decentralized Application Platform*. [On-line], https://github.com/ethereum/wiki/wiki/White-Paper last visit August 26, 2019.

[38] Coinbase. *Why is my transaction pending*. [On-line], https://support.coinbase.com/customer/en/portal/articles/593836-why-is-my-transaction-pending- last visit August 26, 2019.

Abbreviations

ABI	Application Binary Interface
DOM	Document Object Model
DoS	Denial-of-Service
EVM	Ethereum Virtual Machine
GUI	Graphical User Interface
MVC	Model-View-Controller
PoC	Proof-of-Concept
SLA	Service Level Agreement
SLO	Service Level Objective
SP	Service Provider
SC	Smart Contract
TTP	Trusted-Third-Party
UML	Unified Modelling Language
WSLA	Web Service Level Agreement

Glossary

Access Control Restricting access to a certain part of an application to a specified user or group of users.

Rest API An interface of an application allowing systems to make a request to a specific URI to retrieve a resource.

Authentication The act of validating the identity of a user

Authorization Authorization is the decision whether an entity is allowed to perform a particular action or not, *e.g.,* whether a user is allowed to attach to a network or not.

Blockchain An immutable distributed ledger that secures transactions with the use of cryptography principles.

Deployment Deployment is the process of making software available for use.

Ether The cryptocurrency of the Ethereum platform.

Ethereum gas Ethereum introduces the concept of gas, which is a unit that specifies the number of operations that a miner should perform to include a transaction in the blockchain.

Hash Result of hash function often used in cryptography.

High Level Language A programming language that is clearly abstracted from the level of machine language.

Integrity Integrity refers to the quality of the data over its entire lifetime and means that data is consistent, accurate and reliable.

Service Level Agreement (SLA) A contract between two parties that specify what the service provider should deliver in terms of quality but does not define the specific technologies used to provide such a service.

Service Level Objective A term defined in a SLA which specifies the required requirements for the service, such as Quality of Service (QoS) metrics.

Service Credits Used by a Service Provider to compensate a customer in case of a violation of a Service Level Agreement which the customer can use to pay for the upcoming service bills.

Smart Contract An executable code that runs on a blockchain.

Turing completeness A programming language is said to be Turing complete, if it is theoretically capable of performing all computations which a computer could do if it had infinite memory.

WSLA Framework developed by IBM to represent an SLA in an XML-based format.

List of Figures

List of Tables

Appendix A

Installation Guidelines

In order to facilitate the deployment, `SLAMer` is split into 3 docker containers; one for the `database`, one for the `backend` and one for the `frontend`. Further, Ganache is needed to simulate an Ethereum blockchain. Perform the following steps to setup and run `SLAMer`.

A.1 Getting Started

1. Go to `https://www.trufflesuite.com/ganache` and install the Ganache GUI. If using Linux, you might need to install it from github: `https://github.com/trufflesuite/ganache/releases`.

2. Download and install `Docker` from: `https://docs.docker.com/v17.12/install/`. Scroll through the page and select the installation for your OS. For Linux, go to this url: `https://docs.docker.com/install/linux/docker-ce/ubuntu/`

A.2 Ganache Setup

Once Ganache is installed, run it and select "Quickstart" on startup. A list of Ethereum accounts should be displayed, which can be ignored for now. Navigate to the settings (Top right button) and select the "Server" tab.

Select the *Hostname*. It is recommended to use the IP address of your local network, which should be something similar to 192.168.x.x and marked with "Wi-Fi" or "LAN". The *Port* can be left as is.

Disable *Automine* and set a reasonable *Mining Block Time* of around 10 seconds. Click on "Save and Restart" to apply these settings.

A.3 SLAMer Setup

Run SLAMer

Open a terminal window and navigate to the root folder of SLAMer. Then, run the following command:

```
$ docker-compose up
```

This will will take some time until all the dependencies are installed in the containers. Once the message

```
Started BackendApplication in XX seconds
```

is displayed, SLAMer is ready to use.

Configure SLAMer

Open up a browser window and navigate to localhost:3000, where a registration screen will be prompted. For the wallet and private key, the credentials from one of the accounts provided by Ganache can be taken. After registration and login, go to the settings section (Top right). Under "Ganache URL", enter the IP address and port configured in Section A.2 in the following format:

http://*IP*:*Port*

Example: http://192.168.1.118:8545

Click on "Save". SLAMer is now ready and users can create SLAs and deploy SCs on the blockchain in Ganache.

A.4 Setup Monitoring

In order to test the monitoring functionality, one can use a HTTP client, such as Postman. For Postman, go to https://www.getpostman.com/downloads/ and install it.

For now, the SLAMer API only provides one endpoint for verifying an Average Response Time SLO. To verify an Average Response Time SLO, prepare a JSON payload with following fields:

```
1 {
2   "measured": int value of measurement,
3   "sloId": ID of SLO,
4   "slaId": ID of SLA,
5   "wallet": "wallet address of monitoring service"
6 }
```

Listing A.1: Payload

To get the relevant values, navigate in `SLAMer` to the details view of an SLA containing an SLO of type Average Response Time. The *Agreement Number* corresponds to the `slaId`. The ID of the SLO can be found on the *Service Level Objectives* Section at the bottom and the wallet address of the monitoring service is also listed on this page.

Now, perform a POST request with this payload to the following URL:

`http://localhost:8080/monitor`

This payload is sent to blockchain, this is why it will take some time until the HTTP response returns. After 5 attempts with a `measured` value above the value specified in the SLA, the SLA should terminate and the customer compensated.

Appendix B

Contents of the CD

- **BA-Thesis-Carlos-Schweizer.pdf** Final Thesis as pdf.
- **BA-Thesis-Carlos-Schweizer.zip** Final thesis source code (LaTeX).
- **Figures** All the figures as pdf and draw.io.
- **Midterm-Presentation.pptx** Slides of the midterm presentation as pptx.
- **Midterm-Presentation.pdf** Slides of the midterm presentation as pdf.
- **Slamer** Source code (folder)