Enabling Dynamic SLA Compensation Using Blockchain-based Smart Contracts

Eder J. Scheid[1], Bruno B. Rodrigues[1], Lisandro Z. Granville[2], Burkhard Stiller[1]
[1]*Communication Systems Group CSG, - Department of Informatics IfI - University of Zürich UZH*
[scheid,rodrigues,stiller]@ifi.uzh.ch
[2]*Institute of Informatics INF - Federal University of Rio Grande do Sul UFRGS*
granville@inf.ufrgs.br

***Abstract*—Service Level Agreements (SLA) are documents that specify what Service Providers (SP) are delivering to customers. They contain information about the service, such as target performance level or monthly availability, and penalties for the violations of the SLA. The information about the penalties is essential because if the SP does not deliver what is defined, the customer must be compensated accordingly. However, the current compensation process is cumbersome and complex because of the amount of involved manual effort. To address this issue, it is proposed in this paper an approach based on blockchain and Smart Contracts (SC) to automate the compensation process while enabling dynamic payments during the SLA lifetime. The proposed approach was evaluated in an use case that simulates the management of a Quality of Service SLA between an SP and a customer. Based on the performed evaluation, parts of the SLA management process were successfully automated using a decentralized solution, and the payment of the compensation occurred without the intervention of a third party.**

***Index Terms*—SLA Management; Quality of Service; Billing; Blockchain; Smart Contract.**

I. Introduction

Service Level Agreements (SLA) are contracts between Service Providers (SP) and customers specifying what customers can expect from the service but not how that service is delivered [1]. From a network management perspective, an SLA is essential to define service obligations and boundaries, enabling one to check whether a defined service is delivered as contracted. For instance, an SLA can document Quality of Service (QoS) requirements, *e.g.,* VoIP latency $\leq$ 50 ms. However, it does not describe the technology or solution used to deliver such a VoIP service. In addition to QoS requirements, SLAs also include obligations of both parties, *e.g.,* service fee, and compensation value. It is crucial to have these aspects documented in SLAs because, in case of an SLA violation (*e.g.,* when the performance levels are not met), the customer can submit a claim to the SP to be compensated for such a violation [2].

The compensation process, however, is not straightforward because both parties can act dishonestly. For example, if a customer submits a claim that an SLA was violated, it is necessary to provide evidence (data) that supports this claim. In turn, the SP needs to provide evidence that the service was compliant with the SLA. In this sense, both parties can tamper with the data to corroborate their claims. Moreover, the customer may decide not to pay the defined amount for the service, while the SP can decide not to pay the defined compensation. As a possible solution for this situation, the SP and the customers typically rely on a Trusted Third Party (TTP), such as a bank or a financial services company, to ensure the correct payment, which is costly and introduces bureaucracy into the process. Thus, there is a need for a solution that, in addition to dispensing with costly and slow TTPs, ensures that the data, once monitored, cannot be tampered with and that the payment from both sides will be correctly performed.

Blockchains and Smart Contracts (SC) can help address these issues by providing an immutable data storage and a trustful payment system. In this sense, SCs executing in a blockchain can be used to express, in a tamper-proof manner, SLAs containing quantitative terms, such as QoS metrics. Moreover, SCs provide means to automatically exchange funds between network peers in a secure and trusted fashion, thus being a promising technology to enable the payment of services fees and compensation values without the need of a TTP.

In this paper, it is presented the design and implementation of a blockchain-based SC that simplifies and automates the compensation process in case of an SLA violation. With the employment of an SC, both SP and customer have the assurance that the contract will be enforced and that the terms of the SLA will not change. Moreover, the presented approach replaces the TTP and enables a dynamic interaction between SPs and customers. The contributions of this work are:

- Translation of QoS-related SLAs in blockchain-based SCs;
- Guaranteeing a transparent and immutable SLA; and
- Automated SLA compensation and service fee payment.

Further, the presented SC addresses four out of the six phases of the SLA management lifecycle (which is detailed in Section II), including the service fee payment and the trusted SLA monitoring.

The remainder of this paper is structured as follows. In Section II, an overview of concepts involved in this work is provided; and the related works to the approach are described. In Section III, the general approach design, and associated implementation are presented. In Section IV, an evaluation of the SC is performed. This evaluation is discussed in Section V. Finally, in Section VI conclusion and future work directions are presented.

978-3-903176-15-7 © 2019 IFIP

II. Background and Related Work

In this section, a background on *(A)* the SLA management lifecycle, *(B)* the current compensation process, and *(C)* blockchains and smart contracts is provided. Moreover, *(D)* current works relating to the approach are described.

A. SLA Management Lifecycle

The lifecycle of SLA management can be divided into six phases [3]. This cycle restarts every time a new service is contracted or renewed by the customer.

1 - Discover Service Provider – The customer identifies which SP can deliver the necessary resources that meet their needs. Thus, the customer must have a precise definition and scope of the desired service and conduct an investigation to list current service necessities.

2 - Define SLA – This phase often involves a negotiation between SP and customer, in which they negotiate the terms and target performance level for the services that are going to be included in the SLA. It is paramount that both parties agree on the definition of the provided service and related concepts to ensure that the SLA is unambiguously and consistent during the SLA lifespan. If one party does not agree, then the process has to start again, until a consensus is reached.

3 - Establish Agreement – Comprises the definition and development of the template in which the terms defined during Phase #2 are going to be placed. Once both parties sign the contract, they are committed to obligations and penalties.

4 - Monitor SLA Violation – The monitoring of the service is vital to detect if the SP is meeting the defined performance level. If the service is not met, then the customer must be compensated accordingly. Therefore, both parties should actively monitor the service or rely on a TTP monitoring solution. In either case, the monitoring solution must provide reliable measurements, which is a challenge in itself.

5 - Terminate SLA – Depending on the terms defined in Phase #2, the termination of an SLA happens when the SLA validity has expired or when an SLA violation is detected. However, each SLA violation incident is accounted to calculate the payment of the compensation value to the customer.

6 - Enforce Penalties for SLA Violation – If the SP fails to meet any agreed term, then the customer must be compensated. In this phase, the compensation is calculated based on the penalties defined in the SLA, which can be defined in the form of service credits or fiat-currency.

B. Current Compensation Process

The compensation process occurs in the last phase of the SLA management lifecycle (Phase #6), where penalties are enforced if any violation was encountered during SLA monitoring. Taking the Amazon Availability (*i.e.,* Monthly Uptime) SLA [4] as an example, the submission of a new violation claim to receive service credits must follow the predefined steps and be under defined guidelines.

First, the customer must submit a violation claim by opening a case in the AWS Support Center. The Amazon response team must receive this newly opened claim until the end of second billing cycle after the customer has observed the violation. Assuming that the customer has submitted a claim, and the responsible team has received it in time, the AWS team verifies the claim to check whether it contains all the necessary information, such as *(i)* the term "SLA Credit Request" in the subject line, *(ii)* the dates and times of each claimed violation, *(iii)* the affected contracted instances or volumes, and *(iv)* the request logs (with sensitive information redacted) that document and corroborate the claimed violation. If the AWS team acknowledges the claim, then the compensation is paid according to Table I. However, if just a single piece of information is missing, then the team will not acknowledge the claim. Thus, the compensation will not be paid. It is noticeable that this process is manual, complex, and prone to errors.

Monthly Uptime Percentage	**Service Credit Percentage**
Less than 99.99% but $\geq$ 99.0%	10%
Less than 99.0%	30%

TABLE I: Compensation Percentages and Thresholds

C. Blockchain and Smart Contracts

The blockchain concept is relatively recent, being introduced in 2009 with the creation of Bitcoin [5]. A blockchain is a distributed database composed of chained blocks, where each link in the chain is the hash of the previous block header, pointing towards the first block created (aka genesis block). To alter one block, it is necessary to change the hashes of all following blocks and collude the majority of the peers in the network to accept the changes. Once information is appended and distributed in the blockchain network, the effort to change or remove it is exceptionally high. Thus, immutability is achieved by using a combination of hash algorithms and decentralization approaches, providing tamper-proof storage.

The data is replicated among all network peers, which ensures the availability if there is at least a single copy in the network. One problem related to this decentralization is to reach a consensus on the data by peers; this problem is solved by most implementations using a Proof-of-Work (PoW) algorithm. In such an algorithm, peers, called *miners*, validate blocks by generating hashes using a specific protocol; the first *miner* that finds a hash matching a pattern is allowed to append the block and receives a reward (*e.g.,* cryptocurrency).

Taking advantage of the data immutability and decentralization provided by the blockchain, the concept of blockchain-based Smart Contracts (SC) emerged. An SC is an executable code that runs in a distributed manner in the blockchain [6]. The range of applications that can be implemented in such SCs depends on the underlying execution environment that the blockchain provide. An SC can be viewed as a procedure in a relational database management systems, as pointed by [7]. Once an SC is deployed in a blockchain, it will receive a unique address, and all the interaction will be performed by sending transactions to this address. Also, upon receiving a transaction, the SC will automatically execute in every node in the network, following the implemented code, which once deployed, cannot be altered.

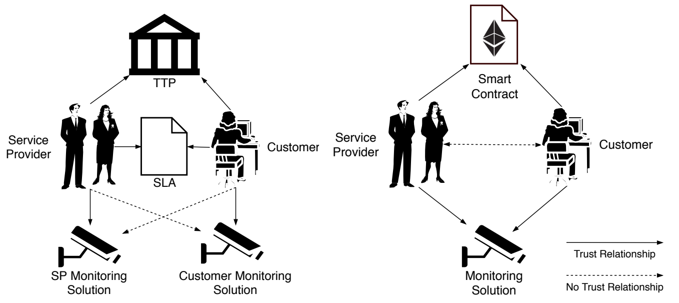
D. Related Work

Due to the novelty of the blockchain technology, there is still a considerable amount of work for it to reach full maturity. This assumption is also valid when talking about blockchain-based SCs. However, even with the infancy of such concepts, some efforts were made regarding their employment in the SLA management area.

The authors of [8] present a formal specification to describe Web APIs and their SLAs. This specification is based on Resource Description Framework (RDF), which is a standard model to exchange data on the Web. Also, it is presented in the paper, an Ethereum SC to manage the contract of such Web APIs by the customer. However, this SC is only focused on Web APIs, and the compensation process is not described. Moreover, in [9] and [10], two proposals to use Ethereum SC in the SLA context are presented. The former proposes to manage Small-Cell-as-a-Service (SCaaS) agreements between the small-cell owners and network operators using an Ethereum SC. The contract registers these owners and periodically informs them about payments (performed by network operators) that can be withdrawn. Still, it focuses on SCaaS and does not delve in the monitoring or compensation aspect. The latter, in turn, proposes to manage cloud billing services and SLA monitoring with Smart Contracts and presents the concept of SLA-coins, which are counted at the end of the month to check whether the SLA was violated or not. However, no implementation details of the SC are presented. Therefore, the feasibility of the approach cannot be evaluated.

III. Approach

This section describes the proposed approach, being divided into two subsections. The first subsection describes the whole design of the SC, providing information, assumptions, and arguments to support the approach. The second subsection details the technical implementation of the SC, reporting employed technologies, programming languages, and challenges. One example of an application of the SC described in this paper can be found in [11], which details its use in NFV environments.



(a) Current Trust Relationship (b) Approach Trust Relationship

Fig. 1: Trust in Different Compensation Scenarios

It is essential to define the role and trust relationship of involved actors before delving into the design. These actors were derived from the SLA management lifecycle; they are the SP, the customer, and the monitoring solution. Figure 1a represents the trust relationship between the actors involved in the current compensation process. Currently, both parties (SP and customer) have to trust the SLA, in the TTP, and in their respective local monitoring solutions. Moreover, the customer might not trust in the monitoring solution provided by the SP because the SP can tamper with it, and the SP might no trust in the customers monitoring solution for the same tampering reason.

Figure 1b depicts the trust relationship of the proposed approach. In one side, there is the SP that can be any company that provides a resource that follows an "*X*-as-a-service" model. The SP is responsible for creating and deploying the SC in the blockchain. On the other side, the customer is the party that buys or requires this resource *X*. The relationship between these is not trusted, as described in Section I. However, there is one relationship (besides the one with the SC) that it is assumed that both parties, SP, and customer, have total trust, which is their relationship with the monitoring solution. The assumption that these parties trust that the monitoring solution reports correct values and actual violations must be made because the SC cannot act outside the blockchain. Thus, the monitoring solution acts as an "oracle" that provides trusted information to the SC.

A. Smart Contract Design Decisions

The design decisions of the SC are divided into four main parts: (1) Content, *i.e.,* what to store in the SC, (2) Authorization, *i.e.,* who is allowed to interact with the SC, (3) Confidentiality, *i.e.,* data visibility, and (4) Accounting/Business Model, *i.e.,* how the fee and compensation payment will work. These parts are described in the following items.

1) **Content** – The description of the basics SLA components provided by [12] guided the selection of which components are going to be implemented in the SC. The components implemented in the SC are: *validity period*, *scope*, *parties*, *Service-Level Objectives (SLO)*, and *penalties*. For the sake of simplicity, it is assumed that one SLA contains only one SLO. However, it is possible to modify the SC to include any number of SLOs in the SLA. For each new SLA specified, a new SC is deployed in the blockchain. This decision allows control over each SC and simplifies the examination of the SC by the customer. Regarding the storage of measurements provided by the monitoring solution, the SC only stores the last violated value and the number of violations during a period of time. The decision of not storing all the measurements was taken because storing data in the blockchain is costly. Thus, storing only the last violated value and the violation count can help to reduce costs related to the monitoring.
2) **Authorization** – The access control of who is allowed to interact with the SC was examined. To maintain this access control, the developed SC stores information about the blockchain addresses of the SP, the customer, and the

monitoring solution. The SP address is included once the SC is deployed in the blockchain and the SP sets the customer address. These addresses cannot be changed during the SC operation. The monitoring solution address is composed of two addresses, one provided by the SP and the other provided by the customer. If these two addresses match, then the monitoring solution is verified; otherwise, then the monitoring cannot interact with the SC. Only this verified monitor address is allowed to send violation notifications to the SC. Therefore, the inclusion of such addresses and the implementation of verifications guarantee that only authorized parties can interact and modify information stored in the SC.

3) **Confidentiality** – According to [13], "*...everything that is inside a (blockchain-based smart) contract is visible to all external observers. Making something (e.g., a variable) private only prevents other contracts from accessing and modifying the information...*". This statement has to be taken into consideration if the SP does not want to disclose information, such as the price of a particular service. This issue can be solved by encrypting this information using encryption methods, such as RSA [14]. In such a method, the data is encrypted using public/private key-pairs, in this case, the price will be encrypted using the public key from the customer, and only the customer holding the private key will be able to decrypt it. Thus, research on employing homomorphic encryption methods might help to prevent unwanted disclosure of information stored in the SC. Moreover, blockchains provide *pseudonymity* as addresses are not related to any real-world information, being just a fixed-size hash; therefore, it is difficult to link an address to a real person or company.

4) **Accounting/Billing Model** – To allow the dynamic compensation to the customer and automatic subscription payment without the need of a TTP, the SC must hold funds to realize these transfers. Therefore, it is expected that the customer sends the subscription fee before the start of the service, and this fee is locked in the SC until the end of the SLA. Once the SLA is finished, the remaining funds are transferred to the SP. This is an unusual business model in cloud services; however, it can be found in "pre-paid" models and real estate renting models, where the customer pays in advance to rent a property for the subsequent period.
The dynamic calculation of the compensation value works, inside the SC, as follows. Assuming that the monitoring solution will perform measurements every $P_{monitoring}$ minutes (the default value is 5 minutes), the sum of the time that the measurement was different than the agreed performance level (*i.e.,* SLA violated) during the interval of compensation ($P_{compensation}$) divided by the total expected measurements ($Total_{measurements}$) characterizes the period of violation ($P_{violated}$), see Equation 1 and 2. This means that the SP was not delivering the defined performance level during $P_{violated}$ (in %) of the compensation interval. The values of $P_{monitoring}$, and $P_{compensation}$ can be customized for each new SC and are used to calculate the compensation value. The compensation value (C_{val}) is calculated using Equation 3. If the $P_{violated}$ did not reach a threshold of α (*e.g.,* 10%), then the customer is not compensated. However, if the $P_{violated}$ is above this α, then the customer receives a fraction of the SLA price corresponding to the compensation period as compensation. In Equation 3 the percentage is fixed at 0.1 to illustrate a percentage; nevertheless, this value is set during the SLA definition. The calculation of the compensation is performed every compensation period to allow the dynamic compensation to the customer.

$$Total_{measurements} = \frac{P_{compensation}}{P_{monitoring}} \tag{1}$$

$$P_{violated} = \frac{\sum violations}{Total_{measurements}} \tag{2}$$

$$C_{val} = \begin{cases} 0, & \text{if } P_{violated} \leq \alpha \\ SLA_{price} \times \dfrac{P_{compensation}}{SLA_{validity}} \times 0.1, & \text{if } P_{violated} > \alpha \end{cases} \tag{3}$$

where

$$\{\alpha, P_{violated} \in \mathbb{R} \mid 0.0 \geq \alpha, P_{violated} \leq 1.0\}$$

B. Implementation Details

The SC is implemented in Solidity, which is a Turing-complete SC language provided by the Ethereum blockchain. This language is influenced by C++, Python, and JavaScript and runs on the Ethereum Virtual Machine (EVM), a VM explicitly designed for the Ethereum blockchain [13]. The choice for such an SC language and blockchain is due to the fact that the calculation of a dynamic compensation requires complex calculations, which cannot be performed using non-Turing complete SC languages, such as the Bitcoin Script language [15]. It worth mentioning that Solidity is a modern programming language, released in 2013; thus, it is still in constant development and presents some security vulnerabilities. One vulnerability is regarding the overflow and underflow in arithmetic operations in SCs [16]. To one's advantage, a library that performs all the necessary verifications to avoid overflows and underflows was developed, called SafeMath [17]. Therefore, SafeMath was employed to perform all the operations, such as multiplications, divisions, sums, and subtractions, using the type `uint`, *i.e.,* unsigned integer.

Regarding the storage of SLA information, a `struct` is implemented containing the **content** described in Section III-A. The defined `struct` is represented in Listing 1; it contains information about the SLA validity period, target performance level, price, compensation value, and relevant information.

```
struct validityPeriod {
    uint createdAt; // timestamp
    uint startTime; // timestamp
    uint endTime; // timestamp
    uint validity; // seconds }
struct slaStruct {
    string service;
    uint performanceLevel;
    string unit;
    uint currentPerfomanceLevel;
    uint price;
    bool paid;
    uint compensationPercentage;
    uint compensationPeriod;
    uint compensationValue;
    uint currentCompensationInterval;
    uint compensationThreshold
    uint monitoringPeriod;
    uint violationCount;
    uint totalExpectedMeasuraments;
    uint compensationValue;
    validityPeriod period; }
slaStruct private currentSla;
```

Listing 1: Solidty Struct Containing SLA Information

The Ethereum public addresses of the SP, customer, and monitoring solution are stored as the type `address` in Solidity (*cf.* Listing 2). This type of variable is peculiar because it was designed to hold Ethereum addresses (a 20 bytes value) and contains two special functions, one to check the account balance (`balance()`), and other to transfer funds to the address (`transfer()`). As the SC handles funds (*i.e.,* ethers), this type was selected to store the addresses.

```
//addresses
address private serviceProviderAddress;
address private customerAddress;
address private monitorSP;
address private monitorCustomer;
```

Listing 2: Addresses Variables

Listing 3 depicts an example of how the SC allows only **authorized** addresses (*cf.* Listing 2) to interact with the SC. For instance, once the SC is deployed, it automatically executes the `construct()` function (line 1), which sets the SP address as the address that deployed the contract. Only the SP is allowed to set the customer address. Therefore, the `setCustomer()` function (line 7) contains the `require()` function (line 8), which verifies whether the sender of the message matches with the stored SP address using the `isSP()` function (line 4). This verification with `require` functions is performed in every SC function that requires access control. There are functions that only the SP can call, and others that only the customer can call, such as the deposit function (*cf.* Listing 6). The `require(condition)` function can revert the transaction if the condition is false.

```
1 constructor() {
2     serviceProviderAddress = msg.sender;
3 }
4 function isSP(address sender) private view returns (bool)
      {
5     return sender == serviceProviderAddress;
6 }
7 function setCustomer(address _customer) public returns (
      bool) {
8     require(isSP(msg.sender));
9     customerAddress = _customer;
10    return true; }
```

Listing 3: SC Constructor and Authorization Functions

Every time the monitoring solution detects a violation, *i.e.,* when the measured value is above the target performance level defined in the SLA, it triggers the `setViolation()` method to inform the event. The function to set this violation is depicted in Listing 4, which is only accessible by the monitoring solution address and requires that this address must be verified (*i.e.,* SP and customer monitoring solution address must match). This function calculates the period violated by dividing the number of detected violations in the interval divided by the total expected measurements in the interval (line 16). If this period of violation is above the defined threshold, *e.g.,* 10% (line 17), and if the compensation interval has ended, then the compensation is **paid** using the compensation function depicted in Listing 6.

Operations and functions that rely on time use the block *timestamp* as a reference for the current time. The block *timestamp* is the time in seconds since the UNIX Epoch (*i.e.,* January, 1st 1970) that the current block has been mined and appended in the blockchain.

```
1 function setViolation(uint _level) public returns (bool) {
2     require(monitorVerified());
3     require(isMonitor(msg.sender));
4
5     if (isTerminated)
6     {
7         return false;
8     }
9     if (currentSla.currentPerfomanceLevel != _level) {
10            currentSla.currentPerfomanceLevel = _level;
11    }
12
13    currentSla.violationCount = currentSla.violationCount
          + 1;
14    if (block.timestamp <= (currentSla.period.startTime +
           (currentSla.compensationPeriod * currentSla.
          currentCompensationInterval))) {
15        //Carry on with normal operation
16     }
17    else {
18        uint periodViolated = ((currentSla.violationCount
              * 100)/ (currentSla.totalExpectedMeasuraments
              ));
19        currentSla.violationCount = 0;
20        if (periodViolated > currentSla.
              compensationThreshold) {
21            compensateCustomer();
22        }
23        currentSla.currentCompensationInterval++;
24    }
25    isSlaFinished();
26    return true;
27 }
```

Listing 4: Set New Violation Function

As the SC is not able to automatically execute a function by itself, the SP, the customer, or the monitoring solution must periodically call the SC to verify if the SLA is still valid or not. This verification is performed using the `isSlaFinished()` function depicted in Listing 5. If the current block *timestamp* is above the calculated end time (start time plus validity), the SC verifies if there is compensation to be paid, if not, it terminates the SLA (line 13), transferring the remaining SC balance to the SP. The `private` keyword after the function arguments specify that the function can only be called from within the contract. Thus, providing another level of trust to involved parties, as no outside interaction is possible.

```
function isSlaFinished() returns (bool) {
    require(isSP(msg.sender) || isCustomer(msg.sender) ||
         isMonitor(msg.sender));
    if (block.timestamp >= currentSla.period.endTime) {
        uint periodViolated = ((currentSla.violationCount
             * 100)/ (currentSla.totalExpectedMeasurements
             ));
        currentSla.violationCount = 0;
        if (periodViolated > currentSla.
             compensationThreshold) {
           compensateCustomer();
        }
        terminateSLA();
        return true;
    }
    return false;
}
function terminateSLA() private {
        isTerminated = true;
        serviceProviderAddress.transfer(address(this).
             balance);
}
```

Listing 5: Check SLA Validity Function

The compensation function (*cf.* Listing 6) is called to transfer the compensation value for the customer address stored by the SP. Having the customer address stored as a variable of the type `address` allows the SC to call the `transfer` function, which is inherited from this variable type, to send the defined compensation value. Another function that relates to the customer is the `deposit` function. This function is public, meaning that anyone can call it. However, the sender must be the customer address provided by the SP and the value transferred to the SC must match with the value defined in the SLA negotiation. A peculiarity of this function is that if everything occurs normally, then the function yields an event of the type `CustomerDeposit` with the customer address and price as arguments using the function `emit`. This event can be monitored by the SP to verify when the contracted resources should be available to the client; an example of such a process can be found in [11]. Thus, the release of resources by the SP can be managed in an automated manner as well.

```
function compensateCustomer() private returns (bool) {
    customerAddress.transfer(
     currentSla.compensationValue
    );
    return true;
}
function deposit() public payable returns (bool) {
    require(isCustomer(msg.sender) &&
        (msg.value == currentSla.price) &&
          !currentSla.paid);
    currentSla.paid = true;
    currentSla.period.startTime = block.timestamp;
    emit CustomerDeposit(msg.sender, msg.value);
    return true;
}
```

Listing 6: Compensate and Deposit Functions

IV. Evaluation

In this section, it is first presented the evaluation scenario in Section IV-A. Then, the deployment and interaction of the SC are described in Section IV-B. Finally, the performed use case evaluation is detailed in Section IV-C.

A. Scenario

To provide a straightforward evaluation, the selected evaluation scenario describes the application of the approach using the response time metric, which is a QoS-related (*e.g.,* latency, throughput, and packet loss) SLA. The metrics in QoS-related SLAs are quantitative, being able to be automatically verified in the SC. Nevertheless, the designed approach can be modified to fit different types of SLAs, such as service availability.

The evaluation scenario was composed of a web server and a client performing periodic requests. A Raspberry Pi model B with a quad-core ARM Cortex-A7 CPU @ 900 MHz, 1 GB of RAM, and a 150 Mbps Wi-Fi dongle hosted an Apache web server, and a Lenovo ThinkPad T430 with a quad-core Intel(R) Core(TM) i7-3520M CPU @ 2.90 GHz, 16 GB of RAM, and a 1 Gbps Ethernet connection performed the requests as the client. The client requested at every 1 s a PHP page that performed random calculations to introduce load in the server. Moreover, after each finished request, the client retrieved the response time of the request and stored this value in a Comma-Separated-Value (CSV) file.

During a 30 minutes interval, 1800 requests were performed and measured ($30\ min \times 60\ s = 1800$); the response times of these requests are depicted in Figure 2. It can be seen in the graph that the majority of the requests were answered in 0.2 s to 0.3 s, and that there are requests with response times over 0.6 s. The average response time for the period was 0.2445 s. Based on these values, the violation threshold was fixed to 0.3 s, which is depicted using a red dashed line in the graph.

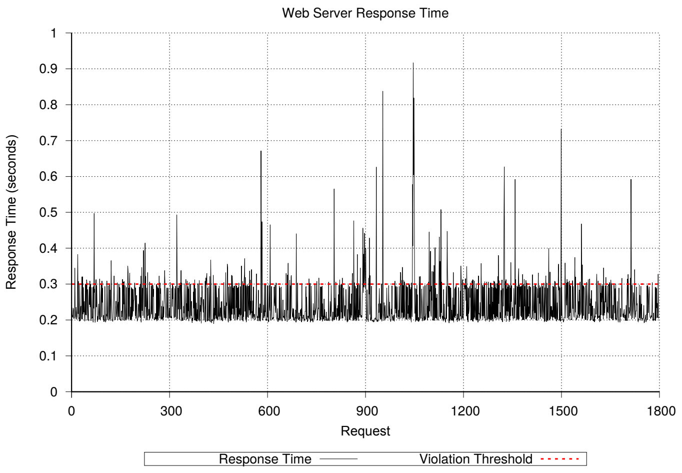


Fig. 2: Response Time between the Client and the Web Server

B. SC Deployment and Interaction

The deployment and evaluation of the SC were performed using Ganache, a permissioned implementation of Ethereum that emulates accounts and funds previously allocated [18]. Moreover, it enables the configuration of blockchain parameters, such as block generation time (which was set to 1 s in the experiment) and the number of accounts. In addition to Ganache, another framework was used to deploy, and perform tests with the SC, called Truffle Suite [19]. Truffle provides a complete framework where it is possible to create automated tests and automatically deploy SCs in the permissioned Ethereum emulated blockchain.

In the Ethereum blockchain, the interaction with SCs is performed using an Application Binary Interface (ABI). This ABI is encoded in a JSON format and defines how the functions and data structures are accessible by other programming languages (*e.g.,* JavaScript using the Web3JS API [20]) outside the blockchain. In addition to the ABI, applications that interact with SCs must know the Ethereum address of the SC. This address is required because it contains the SC code and the data stored within the SC. The Truffle testing suite manages those details for each new SC deployment and tests. In Figure 3 is depicted an example of a transaction receipt for the creation of a new SC in the blockchain. Line 1 contains the hash of the transaction (unique to this transaction), line 2 contains index of the transaction in the block (0 equals the first transaction in the block), line 3 and 4 presents the hash and number of the block that the transaction can be found, and the last line contains the address of the created SC.

```
1 { transactionHash: '0xd75672b3b078c3c6f0562439d40292e98142df62ec0ea5d18a489364f52a0e07',
2  transactionIndex: 0,
3  blockHash: '0xe45ed11e7cf59f7064367599e7a1bc5b173137d036edf68747bb13fb04bb65e5',
4  blockNumber: 5,
5  contractAddress: '0x1b9f5ec63ea1881983d200e265a9e52188a285b8' }
```

Fig. 3: SC Creation Transaction Receipt

C. Evaluation

SLA Parameter	Value
Service Name	"Web Server"
Target Performance Level	0.3
Target Performance Level Unit	"second"
Price	30 ETH
Compensation	30%
Compensation Period	5 min
Compensation Threshold	10%
Monitoring Period	1 s
Validity	30 min
Total Measurements*	300 (per period)
Compensation Value*	1.5 ETH (per period)

* Values calculated automatically by the SC

TABLE II: Smart Contract Parameters

To evaluate the feasibility of the solution, a use case was designed to simulate the management of an SLA using the presented SC. This test works as follows. Firstly, the SC is deployed, and a new SLA is created following the parameters presented in Table II. Secondly, both, SP and customer, inform the monitoring solution address to the SC. Thirdly, the customer deposits the SLA Price in the SC, meaning the start of the SLA. Then, the measurements of the web server response time, described in Section IV-A, are considered as an input of a monitoring agent that performs periodic calls to the SC informing about SLA violations. If the monitored response time is above 0.3 s (arbitrarily defined), then the monitoring agent sends a `setViolation(value)` call to the SC, increasing the number of violations in the period. Once the SC receives this violation, it verifies if the current block *timestamp* is within the compensation interval, if it is not, it calculates the percentage of violations in the period. If this calculated percentage has reached the Compensation Threshold defined in Table II, then it transfers the Compensation Value to the customer. This process is repeated until there are no more values to report. Finally, the SLA is terminated, and the remaining SC balance is transferred to the SP.

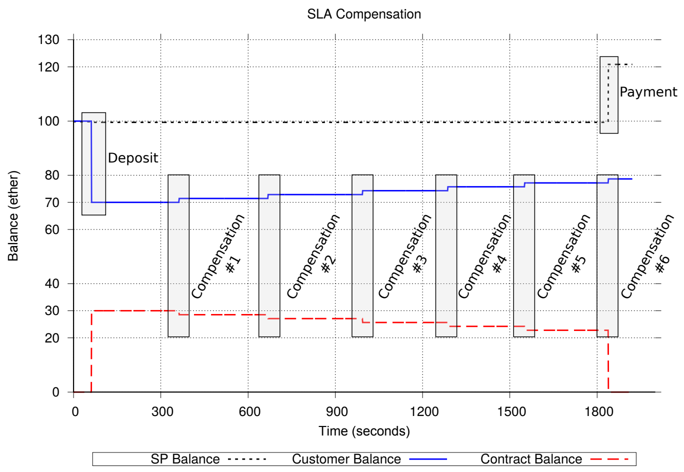


Fig. 4: Stakeholder balances during SLA Validity

Figure 4 depicts the balances of the stakeholders (SP, customer, and SC) during the designed test. The SP balance is illustrated using a dotted black line, the customer balance is represented with a solid blue line, and the balance of the SC is illustrated with a dashed red line. Ganache was configured to allocate 100 ETH to the customer and SP accounts, and the deployment of the SC was performed with the SP account. The first seconds (0 s to 60 s) the contract balance is 0 ETH, but once the customer deposits the 30 ETH corresponding to the SLA price, this balance changes, with the customer balance decreasing to 70 ETH and the contract balance increasing to 30 ETH. Once the deposit occurs, the measured responses times are iterated to verify each one against the Target Performance Level (0.3 s) defined in the SLA. If the value is above the target level, then a violation is reported to the SC. As defined in Table II, the Compensation Period is 5 min (300 s); thus, every 5 min it is expected that the SC calculates if the customer is entitled to compensation or not. Based on the resultant balances, it can be seen that a total of 6 compensations occurred, each one of them during the defined compensation interval. This value of 6 compensations represents the maximum amount of compensations that can occur because the SLA validity was set to 30 min and each compensation occurred every 5 min, resulting in $\frac{30\ min}{5\ min} = 6$ periods. At the end of the SLA validity, the SC performed the payment of the remaining funds (approximately 21 ETH) to the SP and the last compensation (Compensation #6) to the customer.

V. Discussion

The SC proposed in this paper covers the phases #3, #5, and #6, of the SLA management lifecyle. These phases contain elements that can be translated, automated, and stored in an SC, such as the SLA establishment and compensation calculations. They are depicted using a dark gray background in Figure 5. Phase #4 is partially covered because it is assumed that the monitoring solution only reports to the SC violations and not every service measurement. Thus, the SC is only aware of detected violations. This phase is depicted using a light gray background. The proposed SC does not address the two other phases (#1 and #2) because they contain manual human interaction, such as the SLA negotiation and service discovery. These phases are depicted using a white background.

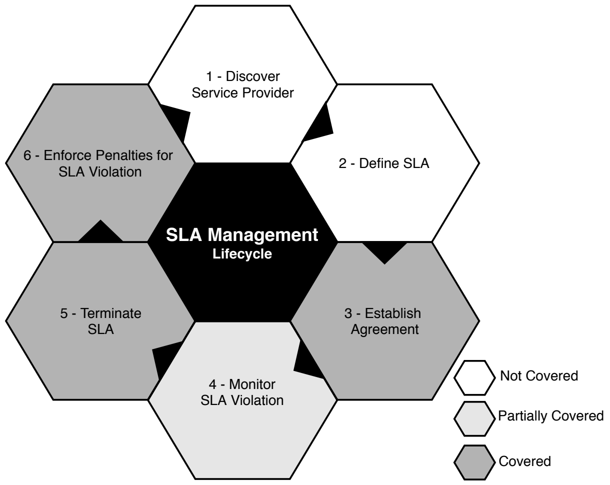


Fig. 5: SLA Management Lifecycle [3]

It is known that most SLAs have a billing cycle of 30 days. Also, the compensation for an SLA violation is often performed using service credits, which are applied in future billing payments [21]. Moreover, it is the responsibility of the customer to submit a claim for an SLA violation and receive this credit. However, with recent advances and popularization of cryptocurrencies, this process can become more dynamic and fast compared to the process described in Section II-B. Cryptocurrencies can be divided into smaller units, for example, 1 ETH can be split into units called *wei*, 1 ETH equals 1×10^{18} *wei*. This division is not possible using fiat currency, such as USD, in which the smallest unit is 0.01 USD. Also, each Ethereum transaction is settled in ˜14 s [22]. Therefore, based on the performed evaluation, it can be seen that the management of stakeholders' (SP and customer) funds was performed entirely by the SC in an automated and dynamic manner. It included the payment of the exact compensation value defined by the SLA to the customer and the transfer of the remaining balance to the SP after the validity of the SLA. Moreover, this process occurred without the intervention of a TTP, which was successfully replaced by a decentralized entity without central control (blockchain-based SC), while providing data immutability, and trusted service and compensation payment.

Concerning the *(i)* scalability, *(ii)* integration, and *(iii)* human intervention aspects of the approach. The first aspect is limited, in theory, to the Ethereum blockchain constraints, such as SC size, and maximum gas used. This is because that one SC is deployed for each SLA between a client and an SP. This type of deployment allows that the management of each SC to be performed outside the blockchain (*i.e.,* off-chain), enabling a scalable solution. If ones desire to integrate the approach with existent systems, adapters to communicate with the Ethereum blockchain must be developed. These adapters must contain functions to interact with the SC, retrieving data and submitting transactions to the SC. Lastly, the third aspect, human intervention, is necessary in complex legal disputes, public relations, and in business strategies. Also, the proposed approach minimizes the manual interaction and errors involved in the process of claiming compensations. Thus, the negotiation of SLA terms and SLA definition still require human interaction.

VI. Conclusion and Future Work

This paper presented a novel approach to address parts of the SLA management process, such as the establishment of the SLA, monitoring, and the enforcement of penalties for violations. More specifically, it focuses on enabling the dynamic SLA compensation in the case of violations during the service duration, *i.e.,* the SLA validity. The presented approach relies on blockchain-based SCs to hold agreed QoS SLA terms by the SP and the customer in tamper-proof storage. Also, the approach automatically manages the SLA billing process, which comprises the payment for the service by the customer and the compensation reimbursement by the SP. The basis for the design decisions of the approach took into consideration the current state-of-the-art in the SLA management lifecycle.

Moreover, it was described the implementation of the approach using Solidity, which is an SC programming language (Turing-complete) provided by the Ethereum blockchain. Listings of the implementation source-code were presented to prove, in a first moment, that is possible to implement the proposed approach. Further, the implemented SC was deployed in a permissioned test Ethereum blockchain to manage an example of a real-life QoS-related SLA (response time). Therefore, based on the performed evaluation, parts of the SLA management process were successfully automated using a decentralized solution and removing the dependency of a TTP to handle the billing process.

There is still a considerable amount of work to provide a production-ready SLA management approach based on blockchain and smart contracts. Future work includes, but are not limited to, *(i)* privacy analysis, *(ii)* cost analysis, *(iii)* scalability analysis, *(iv)* more in-depth research on trusted monitoring solution, and *(v)* integration with existing BSS and OSS. However, it is expected that this work shed light on the current research on the topic.

References

[1] D. C. Verma, "Service Level Agreements on IP Networks," vol. 92, no. 9, Sept 2004, pp. 1382–1388.

[2] O. F. Rana, M. Warnier, T. B. Quillinan, F. Brazier, and D. Cojocarasu, *Managing Violations in Service Level Agreements*. Boston, MA: Springer US, 2008, pp. 349–358.

[3] A. Maarouf, A. Marzouk, and A. Haqiq, "Practical Modeling of the SLA Life Cycle in Cloud Computing," in *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*, Dec 2015, pp. 52–58.

[4] Amazon Web Services, "Amazon Compute Service Level Agreement," Available at https://aws.amazon.com/ec2/sla/ Accessed 23 February, 2018.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Available at http://bitcoin.org/bitcoin.pdf Accessed 08 January, 2018.

[6] M. Alharby and A. van Moorsel, "Blockchain-based Smart Contracts: A Systematic Mapping Study," *CoRR*, vol. abs/1710.06372, 2017.

[7] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[8] H. Nakashima and M. Aoyama, "An Automation Method of SLA Contract of Web APIs and Its Platform Based on Blockchain Concept," in *2017 IEEE International Conference on Cognitive Computing (ICCC)*, June 2017, pp. 32–39.

[9] E. D. Pascale, J. McMenamy, I. Macaluso, and L. Doyle, "Smart Contract SLAs for Dense Small-Cell-as-a-Service," *CoRR*, vol. abs/1703.04502, 2017.

[10] N. Neidhardt, C. Köhler, and M. Nüttgens, "Cloud Service Billing and Service Level Agreement Monitoring based on Blockchain," in *EMISA*, 2018, pp. 65–69.

[11] E. J. Scheid and B. Stiller, "Leveraging Smart Contracts for Automatic SLA Compensation - The Case of NFV Environment," in *IFIP AIMS*. Munich, Germany: IEEE, Jun 2018, pp. 70–74. [Online]. Available: https://files.ifi.uzh.ch/CSG/staff/scheid/extern/publications/AIMS2018.pdf

[12] L.-j. Jin, V. Machiraju, and A. Sahai, "Analysis on Service Level Agreement of Web Services," Jun 2002, technical Report HPL-2002-180, Software Technology Laboratories, HP Laboratories.

[13] Ethereum, "Solidity Documentation," Available at https://solidity.readthedocs.io Accessed 10 July, 2018.

[14] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," vol. 21, no. 2. New York, NY, USA: ACM, Feb 1978, pp. 120–126.

[15] Bitcoin, "Script - Bitcoin Wiki," Available at https://en.bitcoin.it/wiki/Script Accessed 10 July, 2018.

[16] NCC Group, "Decentralized Application Security Project," Available athttps://dasp.co/ Accessed 27 June, 2018.

[17] OpenZeppelin, "SafeMath," Available at https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol Accessed 10 July, 2018.

[18] CONSENSYS, "Ganache — Truffle Suite," Available at https://truffleframework.com/ganache Accessed 31 July, 2018.

[19] ——, "Truffle Suite - Your Ethereum Swiss Army Knife," Available at https://truffleframework.com/ Accessed 31 July, 2018.

[20] Ethereum, "Web3 JavaScript API," Available at https://github.com/ethereum/wiki/wiki/JavaScript-API Accessed 31 July, 2018.

[21] L. Wu and R. Buyya, "Service Level Agreement (SLA) in Utility Computing Systems," *CoRR*, vol. abs/1010.2881, 2010. [Online]. Available: http://arxiv.org/abs/1010.2881

[22] Etherscan 2018 (C), "Ethereum Average BlockTime Chart," Available at https://etherscan.io/chart/blocktime Accessed 23 November, 2018.