Trustworthy Cloud Service Level Agreement Enforcement with Blockchain based Smart Contract

Huan Zhou*‡, Cees de Laat* and Zhiming Zhao*
*Informatics Institute, University of Amsterdam, Amsterdam, Netherlands
‡School of Computer Science, National University of Defense Technology, Changsha, China
Email: {h.zhou, delaat, z.zhao}@uva.nl

***Abstract*—Cloud Service Level Agreement (SLA) is challengeable due to lacking a trustworthy platform. This paper presents a witness model to credibly enforce the cloud service level agreement. Through introducing the witness role and using the blockchain based smart contract, we solve the trust issues about who can detect the service violation, how the violation is confirmed and the compensation is guaranteed. In this model, a verifiable consensus sortition algorithm proposed by us is firstly leveraged to select independent witnesses to form a witness committee. They are responsible for a specific service level agreement and get paid by monitoring and detecting service violation. Through carefully designing the witness' payoff function in the agreement, we further leverage game theory to analyze and prove that it is not the witness itself is trustworthy. Instead, the witness has to tell the truth because of its greedy nature, which is the desire to maximize its own revenue. As long as the service violation is confirmed by the witness committee, the compensation is automatically transferred to the customer by the smart contract. Finally, we implement a proof-of-concept prototype with the smart contract of Ethereum blockchain. It demonstrates the feasibility of our model.**

***Index Terms*—service level agreement, cloud computing, smart contract, blockchain.**

I. Introduction

Traditionally, SLA (Service Level Agreement) is a business concept. It defines contractual financial agreements between roles who engage in the business activity [1]. In the context of cloud computing, it is an agreement between the cloud customer and provider on the quality of the cloud service. Even planning algorithms, like [2][3], try to make the infrastructure satisfy QoS (Quality of Service) requirements of the application, the possibility of violation still exists due to the cloud uncertainty. According to SLA, if the service quality is not met, the cloud provider must compensate the customer's lost. Since a lot of applications tend to run on clouds [4], it is therefore become an important way to ensure their QoS through enforcing SLA, especially when migrating time-critical applications [5][6][7]. Though there exist various of frameworks for service monitoring to detect the service violation [8], the agreement is still hard to be enforced in practice. Because following gaps hinder the traditional SLA to be really adopted and feasible in industry: i) Manual verification. It lacks an automatic mechanism to enforce the agreement, especially for the compensation; ii) Rights fairness. The provider has many more rights, such as the right to verify the violation and decides whether to compensate the customer. iii) Proof of violation. It is hard for the customer to prove and convince the provider that the violation has really happened.

In order to address this issue, there are plenty of research on the cloud SLA. However, most of them concentrate on the syntax definition of the SLA terms and parameters [1], such as SLA*, SLAC, CSLA, etc. Moreover, a systematic survey [1] on cloud SLA has pointed out, that papers they found addressing the issues about SLA violation management and reporting are only 3% and 1% respectively. It demonstrates that detecting the SLA violation and automating the compensation are still challengeable based on traditional technologies.

Smart contract is proposed to digitally facilitates, verifies and enforces a contract through a computer protocol [9]. Some explorations such as [10] combine this concept with cloud SLA negotiation, focusing on the semantic expression of smart contract to automate the negotiation phase. However, most of them lack of a trustworthy underlying platform to execute the smart contract. Blockchain technology brings in a new hint for possible solution to mitigate this gap. Especially the smart contract in Ethereum [11] makes it possible automate the SLA on the blockchain. For instance, [12] designs a set of web APIs based on Ethereum to automate the SLA enforcement. They introduce a role, "Service Performance Monitor", who detects the violation and sends the notification. However, they have not discussed the trust issue for this role. Actually, this is also a challenge for blockchain itself. That is how we can achieve consensus on an event happening outside the blockchain.

In this paper, a witness model is proposed to tackle the challenge of detecting SLA violations in a trustworthy way. A new role termed as *witness* is added in the traditional cloud service delivering scenario to perform as the performance monitor. The witness is designed as an anonymous participant in the system, who desires to gain revenue through offering the violation reporting service. The payoff function for different actions in our agreement model is carefully designed that the witness would have to always behave honestly in order to gain the maximum profit for himself, which can be proved by game theory. In addition, a verifiable consensus sortition algorithm is developed in our witness model to select a certain number of witnesses in order to form a committee. The committee members are randomly selected and the randomness can not be dominated by any participant. This is very important to ensure selected witnesses are independent. Last but not least, a prototype system[1] using smart contracts of the Ethereum blockchain is implemented to automate the SLA lifecycle and empowers the fairness between roles, especially for the customer.

II. SLA Enforcement System and Witness Model

In this section, we first introduce the related roles in our witness model design, especially the witness role. Then we illustrate our system architecture for SLA enforcement with smart contracts on blockchain. At last, we describe our witness model design.

A. Roles and Problem Assumptions

In the traditional cloud SLA lifecycle, there are basically two roles. One is the cloud provider, P, which offers cloud service. The other is the cloud customer, C, which consumes the cloud service and pay the service fee. To demonstrate the key contribution of our work, we take a basic example to formulate our problem as follows.

A cloud provider, p, is an IaaS (Infrastructure-as-a-Service) provider. It provisions VMs (Virtual Machine) on demand with public addresses for its customers to use. For instance, according to the request of a customer c, provider p provisions a VM with a public IP address, IP_{pub}. During the service time, $T_{service}$, the customer, only the customer c is able to SSH and login to the VM through the corresponding address IP_{pub}. In this case, the SLA can be that the provider p claims that during the service time the provisioned VM will always be accessible. If this is true, the customer c must pay the service fee, $F_{service}$, to the provider p after the ending of the service. Otherwise, the customer c can acquire a compensation fee, $F_{compensation}$. That is the customer c only needs to pay $F_{service} - F_{compensation}$ to the provider p in the end, where we assume that $F_{service} > F_{compensation}$. For the latter case, if it happens, we define it as a SLA violation event. In addition, it is worth to mention that we should exclude the case that the inaccessibility is caused by the customer's own network problem, to be a violation event.

With only these two roles in the agreement, it is hard to ensure that the provider can get paid or the customer can get compensation paid back, if the service fee is prepaid. Hence, we leverage blockchain to play as the trusted party to afford a platform for these two roles and enforce these monetary transmissions. But it is still especially difficult to convince both roles whether the violation happens and whether it is caused by the customer's own network problem. We therefore bring in another new role in the traditional SLA lifecycle, named as witness role, W. They are also the normal participants in the blockchain and volunteers to take part in our SLA system to gain their own revenue through offering monitoring service. In order to solve the trust issue, a set of N witnesses, $\{w_1, w_2, ..., w_N\}$, is selected to form a committee in a specific SLA lifecycle. They together report the violation event and may obtain witness fee, $F_{witness}$, as rewards from both the provider and the customer. Moreover, the wallet address of a specific role on the blockchain is denoted by function, $address()$. For instance, $address(w_k)$ is the wallet address of witness w_k.

In this paper, we make the basic assumption on the witness role that it is always selfish and aims at maximizing its own revenue.

B. System Architecture and Model Design

Figure 1 illustrates the system architecture we design for cloud SLA enforcement. First of all, any user of the blockchain, who has a wallet address, can register in the system to be a member of witnesses. They form a witness pool and wait to be selected for some specific contract. The incentive for the witness to participant in this system is to obtain revenue. And the more witness participants in the system, the more reliable and trustworthy the system will be.

The entire SLA lifecycle then becomes as follows. Prior to setting up SLA, the customer c should negotiate with the provider p about the detailed SLA terms, including $T_{service}$, $F_{service}$, $F_{compensation}$, etc. Thereinto, one of the most important terms is to determine N, which is the number of witnesses would be hired for enforcing this SLA. The more witnesses involved in a SLA, the more trustworthy the violation detection results are. On the other hand, however, the more witness fee would be paid and both the customer and the provider need to afford this fee equally. As long as these terms are agreed by both sides, a set of N witness members can be selected to form a witness committee through the sortition algorithm in Section III-A. We design the sortition to be random and being able to convince both, c and p, that most ones in the witness committee are independent and would not belong to

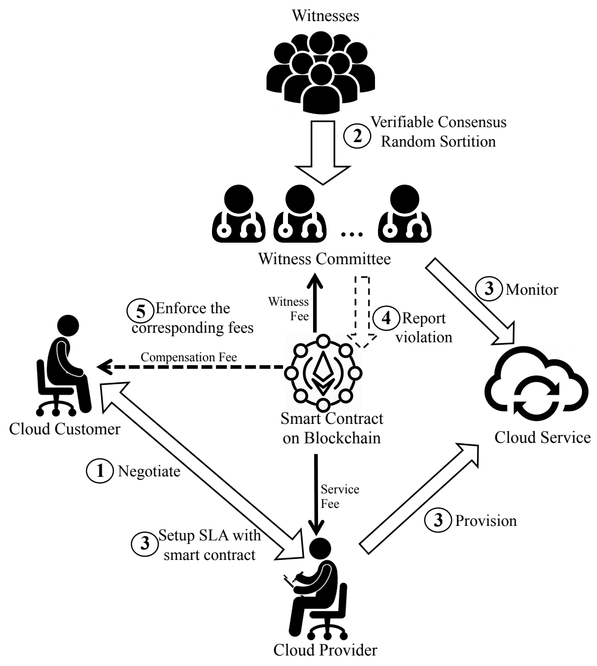

Fig. 1: System Architecture for Cloud SLA Enforcement

[1]https://github.com/zh9314/SmartContract4SLA

the adverse role. After dynamically building up the witness committee, a smart contract can be automatically generated and deployed on the blockchain. Hence, the provider and customer are able to setup SLA through the deployed smart contract. In the mean time, the provider provision its cloud service for the customer to use, and the witnesses from the committee also start to monitor the service. In the case of our problem assumption in Section II-A, the provider p provisions a VM on demand and notify the public address IP_{pub} to all the committee members and the customer c. Therefore, the customer is able to use the VM and each witness starts to "ping" the address IP_{pub} constantly. If the violation happens during the service time, i.e. the address IP_{pub} is not accessible, the witness can report this event independently.

Since the first violation report, the smart contract would start counting a time window, T_{report}. Within this time window, the smart contract accept reports from other witnesses. When the time window T_{report} is over, the violation is automatically confirmed, if there are no less than M out of N reports from the witness committee received by the smart contract. M is also negotiated by p and c. It is then defined in the SLA smart contract. Of course, the bigger the M is, the more trustworthy the violation is. Furthermore, we design that the witness needs to report the violation along with some fee to the SLA smart contract. This is to endorse its report. In some sense, these N independent witnesses constitute a n-$player$ game, in which each witness would like to maximize its revenue. We specially design the payoff function, shown in Section III-B, and leverage the Nash Equilibrium of Game Theory to prove that the witness have to be a honest player in this game. That is they have to report the violation according to the real event.

Finally, the SLA ends at two cases. One case is the service time $T_{service}$ is over and there is no violation. The other case is that the SLA is violated. According to these different cases, the three roles are able to withdraw corresponding fees from the SLA smart contract. This is explained in detail in Section III-B. All the dash lines in Figure 1 mean it may happen according to the actual event. Anyhow, the provider and witnesses from the committee are able to get corresponding fees.

III. Key Techniques

In this section, we describe key techniques adopted in our SLA system in detail. They ensure the automatic detection of the SLA violation can convince both sides, the provider and the customer. First, the verifiable consensus sortition algorithm is leveraged to guarantee that most of the witnesses selected into the committee are random and independent. It is also important to make both sides achieve consensus that most of the selected witnesses would not delegate the opponent's benefit. Based on this, we design the payoff function for the witness model in Section III-B. And also through the Nash Equilibrium theory, we prove that the "player" from the witness committee have to behave honestly and tell the truth to maximize its revenue. In addition, the history of all the actions on the blockchain is immutable. We therefore introduce an auditing mechanism to detect malicious witness.

A. Verifiable Consensus Sortition

In Section II-A, we have explained that the witness role is also an independent normal blockchain participant. It registers itself with its wallet address in our SLA system to be a witness. Its incentive is to gain the witness fee. In order to be a member of a specific SLA witness committee, the registered witness in the system cannot actively choose to participant certain SLA. It can only keep online and wait to be selected randomly. Moreover, neither the provider nor the customer should have the ability to dominate the selection result. Otherwise, most of the selected witnesses cannot be guaranteed not to be in collusion with each other or on behalf of either side in a specific SLA enforcement lifecycle. In addition, the independency among the selected witnesses is also the fundamental requirement to analyze their behavior through game theory in Section III-B. Therefore, a verifiable consensus sortition algorithm is designed as shown in Algorithm 1.

According to the blockchain technique, the participant's wallet address is a fixed length of bits, denoted as $addrlen$. For instance, the Ethereum wallet address is the last 160 bits of the account's public key. Therefore, the address can be interpreted as an integer. We represent the registered witness as an integer set RW. The addresses in the set are listed in ascending order. RW is expressed as Equation 1 and 2.

Algorithm 1 Verifiable Consensus Sortition

```
Input:
    Registered witness set, $RW$;
    Required number, $N$, of members in witness committee;
    Random number, $rand_p$, given by the provider $p$;
    Random number, $rand_c$, given by the customer $c$.
Output:
    Selected witness set, $SW$, to form a committee.
1:  $rand_{hash} \leftarrow Hash(rand_p + rand_c)$
2:  $j \leftarrow 0$
3:  $i \leftarrow 1$
4:  $SW \leftarrow \emptyset$
5:  for all $w_i$ such that $w_i \in RW$ do
6:      if $addr(w_i) > rand_{hash}$ && $w_i$ is online then
7:          add $w_i \Rightarrow SW$
8:          $j++$
9:      end if
10:     if j == N then
11:         break;
12:     end if
13:     $i++$
14:     $i \leftarrow (i \bmod (\|RW\| + 1)) + 1$
15: end for
16: return $SW$
```

$$RW = \{addr(w_k) | addr(w_k) \in (0, 2^{addrlen} - 1)\} \quad (1)$$

$$\forall w_i, w_j \in RW, i < j, addr(w_i) < addr(w_j) \quad (2)$$

To select N witnesses from RW, the provider and the customer must first provide two random numbers, $rand_p$ and $rand_c$ respectively. Then hash function, $Hash(rand_p + rand_c)$, is adopted to generate a hash value, $rand_{hash}$, with $addrlen$ bits. Finally, N online witnesses are selected, whose wallet addresses are bigger and closest to the value $rand_{hash}$. As this value is generated based on the two random values, the set of selected witnesses cannot be predicted in advance. In addition, the result is verifiable by both the provider and the customer. They hence can be convinced that the sortition result is not dominated by the opponent or any other third party. Because they both provide part of the randomness. Though the provider or the customer may exploit several addresses to register as witnesses in the system to delegate its own profit, it is still a small chance that most of its fraudulent witnesses are selected in a specific SLA at the same time, if there are many registered witnesses in RW. There is also an order problem, which is who gives the random number first. The one who gives its random number latter has a small advantage to determine the final selection results. In order to solve this problem, we adopt the commitment scheme [13] mechanism combined with the blockchain based smart contract to make both roles reveal their random values at the same time. Meanwhile, considering the registered witnesses number should be much larger than the witness number, which is $\|RW\| \gg N$, Algorithm 1 can always return the result, SW.

B. Witness Game and Payoff Function

The N witnesses committee for enforcing a specific SLA formulate a witness game. In this game, every witness is equal. Each of them has two actions: report the violation to the smart contract during the service time or keep silence. Here, an endorsement fee is required from the witness when it wants to report the violation. We then design the payoff function as follows. If the violation is finally confirmed, i.e., most of the witnesses report, the ones who have reported gain 10 shares of profit. In this case, the ones who have not reported gain nothing; If the violation is not confirmed, the ones who have reported would not retrieve back their endorsement fee, which is -1 share of profit, as penalty. The ones who have not reported earn 1 share of profit.

The confirmation of the violation is also mentioned in Section II-B. Only when more than M witnesses from the N-witness committee report the service violation event within a time window, the violation is then automatically confirmed by the smart contract. At the same time, the compensation fee for the customer and witness fee are also automatically assigned by the smart contract according to the payoff function mentioned above. The concrete M and N values can also be negotiated to define by the provider and the customer. It is a trade off between the witness hiring expenses and the extent of trustworthy. But in general, the constraints on these values should be $N > 2$ and $M < N$, in order to achieve the violation confirmation reliably and fairly.

TABLE I: Payoff Functions of a 3-witness Game

w_1	w_3			
	$\sigma_3^{(r)}$: Report		$\sigma_3^{(s)}$: Silence	
	w_2		w_2	
	$\sigma_2^{(r)}$: Report	$\sigma_2^{(s)}$: Silence	$\sigma_2^{(r)}$: Report	$\sigma_2^{(s)}$: Silence
$\sigma_1^{(r)}$: Report	(10, 10, 10)	(10, 0, 10)	(10, 10, 0)	(-1, 1, 1)
$\sigma_1^{(s)}$: Silence	(0, 10, 10)	(1, 1, -1)	(1, -1, 1)	(1, 1, 1)

Based on game theory [14], we can prove the Nash equilibrium points in our N-witness game under the given payoff function design. Then, the behavior of the witness is analyzed in order to pursue the Nash equilibrium. However, due to the space, the complete prove is not presented in this paper. Instead, we take the example of the 3-witness game as an example to analyze. In this case, $M = 2$ and $N = 3$. It means that as long as two of three witnesses report, the violation event is confirmed by the smart contract. Therefore, Table I shows the payoff functions of a 3-witness game as an example. Here, w_k represents the kth witness. The vector of the payoff function value is also listed in the witness order. According to this table, it is obvious that the Nash equilibrium points in this game are (10, 10, 10) and (1, 1, 1) respectively. Actually, in a N-witness game, the Nash equilibrium points are still the strategy profiles, where all the witnesses report or keep silence. Based on the property of Nash equilibrium, none of the witness has the incentive to leave the point, as long as it can be achieved. However, neither of these two actions dominates the other action. It means none of the witness can always choose one action to consistently gain the maximum revenue.

Based on above analysis, for a rational and selfish witness, who wants to maximize its revenue through offering services, would have to behave as follows in this game. If there is a violation happening, the witness knows that most of other witnesses are more likely to report this event to gain more revenue. Hence, the higher revenue pushes the witness to report this event. On the contrary, if there is no violation, the witness knows that most of other witnesses are more likely to keep silence. Although the witness wants to achieve the highest revenue, it has to take a great risk to pay a penalty for its fraudulent behavior. From the global view, when there is no violation, all the witnesses prefer to keep silence in order to stay at the Nash equilibrium point, (1, 1, 1). Then the violation acts as a signal to push them achieving another Nash equilibrium point, (10, 10, 10), with much higher revenue. At the same time, they tell the truth about the service violation. Therefore, it is not the witness wants to tell the truth. Instead, it has to be honest, in order to maximize its revenue.

For unrational and malicious witness, we introduce an extra auditing mechanism to kick them off from the witness pool. Because all behaviors on the blockchain are public and immutable. It is possible to audit every witness' behavior history. When some possible malicious behavior is performed by the witness, its reputation value may decrease. As long as its reputation value is not enough, the witness will be blocked by the sortition algorithm to be selected into a witness committee. Then, the witness loses the chance to gain revenue in the system. Furthermore, we can also category some types of possible dishonest witnesses according to certain kind of behavior pattern.

IV. Prototype Implementation

According to the witness model and the payoff function design, we implement a prototype system based on the smart contracts of Ethereum. The language, Solidity[2], of Ethereum is leveraged to program smart contracts.

A. Interactions among Roles and Smart Contract

The sequential diagram in Figure 2 shows how different roles interact with the smart contract especially involving the witness in our model. After witnesses being selected, the entire lifecycle of a specific SLA begins. The provider p provisions the cloud service and deploy the smart contract on the blockchain. In order to setup a SLA, p must prepay the corresponding fee $PF_{prepaid}$ to the smart contract first. The amount of $PF_{prepaid}$ is determined by the half of the maximum witness fee. The customer c is then notified about the service and the content of the smart contract. As all the smart contract on the blockchain is public, the customer can verify the contract and the service status to decide whether to accept the SLA in a certain time window. In order to accept the SLA, the customer also needs to transfer the prepaid fee, $CF_{prepaid}$. It includes the service fee, $F_{service}$ and the other half of the maximum witness fee. As we assume $F_{service} > F_{compensation}$, the compensation fee would be directly deducted from this part of prepaid fee, if the violation happens. Afterwards, every witness in the committee is notified to start monitoring the service continuously.

During the service time, the witness can decide whether to report the event to the smart contract, if there is a service violation, for instance, the VM is not accessible. We design the rule that the witness w_k also needs to transfer a small amount of fee, $WF_{prepaid}$, to endorse its report at the same time. The incentive persuading w_k to report the event is that it would gain relative more revenue as witness fee, if the violation event is finally confirmed by the smart contract. On the contrary, if the violation is not confirmed, w_k would not get back the prepaid endorsement fee, $WF_{prepaid}$, as a penalty. This prevents w_k reporting fake violations just for maximizing its revenue. In addition, the final violation is confirmed according to Section III-B.

B. SLA State Transition

Figure 3 shows the state transition of SLA lifecycle in our smart contract implementation. The reason of designing these states is to restrict the participant's behavior. That is only the allowed roles can interact with the smart contract in a specific state. This is important for finally assigning proper amounts of revenue to corresponding roles in the end of SLA. Meanwhile, we adopt the event mechanism of smart contract to emit an

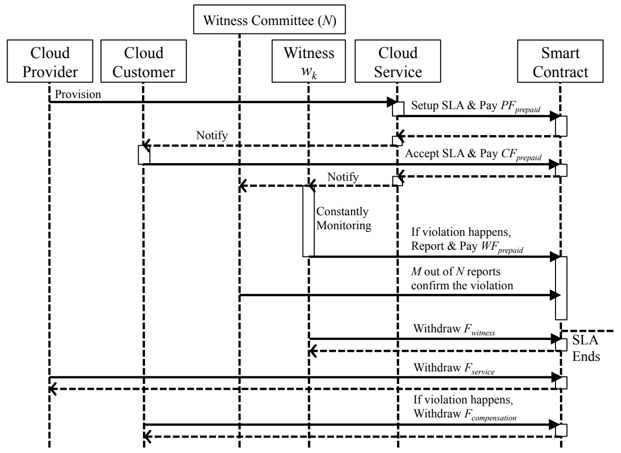
Witness Committee (N)
Cloud Provider
Cloud Customer
Witness w_k
Cloud Service
Smart Contract
Provision
Setup SLA & Pay $PF_{prepaid}$
Notify
Accept SLA & Pay $CF_{prepaid}$
Notify
Constantly Monitoring
If violation happens, Report & Pay $WF_{prepaid}$
M out of N reports confirm the violation
Withdraw $F_{witness}$
SLA Ends
Withdraw $F_{service}$
If violation happens, Withdraw $F_{compensation}$

Fig. 2: Sequential Diagram of Different Roles in Witness Model

event, as long as the SLA state is modified. This event can be leveraged to notify other roles to further invoke its interface and perform corresponding reactions. Because Ethereum also provides the API for off-chain applications to capture these events.

There are five states shown in Figure 3, which are "Fresh", "Init", "Active", "Violated" and "Completed". The arrows in this figure explain how the states transit among each other. The text on the arrow refers to the interface defined in the smart contract, which can be invoked to achieve the state transition. The format of the text is '$role :: interface\_name$'. It means that only the $role$ can invoke the interface, named as $interface\_name$, to make the state transited. The dash arrows demonstrate the state transition path when violation happens. The three squares in the figure represent the corresponding roles in this smart contract. In the end of SLA, they can withdraw the revenue respectively. The text in the box refers to the corresponding roles, who can withdraw revenue from the smart contract under different circumstances. The dash line here also refers to the action adopted under the situation of violation.

The contract is generated in the state of "Fresh". In this state, the provider is able to customize the SLA parameters

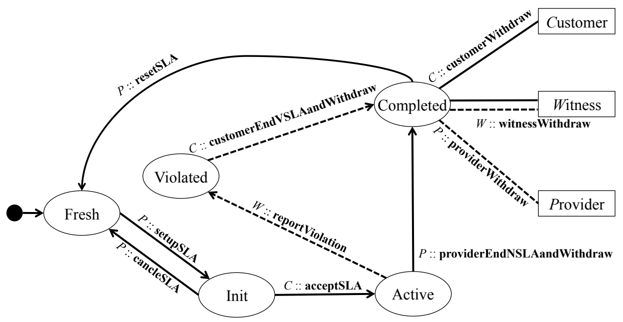

Fig. 3: State Transition Diagram of SLA Lifecycle in the Prototype Smart Contract

[2]http://solidity.readthedocs.io

according to the negotiated results with the customer. Furthermore, the service detail can also be published onto the contract through "publishService". In our case, the detail is the public IP of the VM. All others are therefore being notified. However, this SLA proceeds only when the number of the members in the witness committee is satisfied. Otherwise, the provider is unable to leverage the interface, "setupSLA", to transit into "Init" state. According to the witness model design in Section II-B, the provider needs to prepay some fee, $PF_{prepaid}$, to the smart contract for hiring witnesses. The concrete amount of fee is calculated by the smart contract according to the scale of committee member and a basic hiring fee. In addition, this amount is one of the constraints to invoke the interface. It ensures that only that amount of prepaid fee is transferred into the smart contract. The customer then decides whether to accept. If it accepts the SLA, it also needs to prepay the fee, $CF_{prepaid}$, including the service fee and its part of hiring fee for witnesses. If not, the provider can withdraw back its money and "cancleSLA". When the service is completed, all corresponding roles can retrieve its revenue through a set of withdraw interfaces. After all the money is withdrawn from the contract, the provider can leverage "resetSLA" to rotate back to the previous state. This is used for continuous service delivery instead of a long service duration to stuck witnesses.

It is also worth to mention that the smart contract on blockchain cannot run itself. The state transition must be triggered by some interfaces and it takes some cost to execute. Therefore, we design the interface for the role, who is the greatest beneficiary in some cases, to modify the state. Because they have the motivation to perform the state transition. For example, when the service duration ends normally, the provider is the greatest beneficiary to gain the entire service fee. It must actively leverage the interface, "providerEndNSLAandWithdraw", to end the normal SLA and withdraw its own revenue. Meanwhile, it divides the prepaid money as the payoff function design in Section III-B to different witnesses. Afterwards, other roles are able to withdraw their parts of revenue. Analogously, when there is a violation, the customer is the most motivated one to gain the compensation fee. It can leverage, "customerEndVSLAandWithdraw", to end the violated SLA and transit the state from "Violated" to "Completed".

V. Conclusion

We propose a witness model for cloud SLA enforcement and specially design the payoff functions for each witness. We leverage the game theory to analysis that the witness has to offer honest monitoring service in order to maximize its own revenue. Finally, a proof-of-concept prototype is implemented with the smart contract of Ethereum and is able to demonstrate the feasibility of our model. Via this way, the trust problem is transferred into economic issues. It is not the witness itself would like to be honest, but the economic principles force them to tell the truth. We also believe our witness model based on blockchain can be applied in other scenarios, where originally only two roles are involved in a contract. For the future work, we are going to implement the whole system and combined with our cloud Application DevOps framework, CloudsStorm[3][15], to construct the witness ecosystem. The vision is to insure the cloud performance for applications through automated and trustworthy SLA.

Acknowledgment

This research is funded by the EU Horizon 2020 research and innovation program under grant agreements 643963 (SWITCH project), 654182 (ENVRIPLUS project) and 676247 (VRE4EIC project).

References

[1] F. Faniyi and R. Bahsoon, "A systematic review of service level management in the cloud," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 43, 2016.

[2] J. Wang, A. Taal, P. Martin, Y. Hu, H. Zhou, J. Pang, C. de Laat, and Z. Zhao, "Planning virtual infrastructures for time critical applications with multiple deadline constraints," *Future Generation Computer Systems*, vol. 75, pp. 365–375, 2017.

[3] X. Wang, C. S. Yeo, R. Buyya, and J. Su, "Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm," *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1124–1134, 2011.

[4] K. Jeferry, G. Kousiouris, D. Kyriazis, J. Altmann, A. Ciuffoletti, I. Maglogiannis, P. Nesi, B. Suzic, and Z. Zhao, "Challenges emerging from future cloud application scenarios," *Procedia Computer Science*, vol. 68, pp. 227 – 237, 2015, 1st International Conference on Cloud Forward: From Distributed to Complete Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915030835

[5] Z. Zhao, D. van Albada, and P. Sloot, "Agent-based flow control for hla components," *SIMULATION*, vol. 81, no. 7, pp. 487–501, 2005.

[6] Z. Zhao, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, F. J. Hidalgo, G. Suciu, A. Ulisses, P. Ferreira, and C. d. Laat, "A software workbench for interactive, time critical and highly self-adaptive cloud applications (switch)," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1181–1184.

[7] Y. Hu, H. Zhou, C. de Laat, and Z. Zhao, "Ecsched: Efficient container scheduling on heterogeneous clusters," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 365–377.

[8] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19 – 38, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412121730256X

[9] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *arXiv preprint arXiv:1608.00771*, 2016.

[10] V. Scoca, R. B. Uriarte, and R. De Nicola, "Smart contract negotiation in cloud computing," in *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*. IEEE, 2017, pp. 592–599.

[11] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, 2014.

[12] H. Nakashima and M. Aoyama, "An automation method of sla contract of web apis and its platform based on blockchain concept," in *Cognitive Computing (ICCC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 32–39.

[13] A. Juels and M. Wattenberg, "A fuzzy commitment scheme," in *Proceedings of the 6th ACM conference on Computer and communications security*. ACM, 1999, pp. 28–36.

[14] K. Binmore, *Game theory: a very short introduction*. Oxford University Press, 2007, vol. 173.

[15] H. Zhou, Y. Hu, J. Su, C. de Laat, and Z. Zhao, "Cloudsstorm: An application-driven framework to enhance the programmability and controllability of cloud virtual infrastructures," in *International Conference on Cloud Computing*. Springer, 2018, pp. 265–280.

[3]https://cloudsstorm.github.io/