TU Delft

A Smart Contract for Cloud Service Level Agreement based on Ethereum

University of Amsterdam

System and Network Engineering Lab

Advanced Blockchain Engineering

Author:
Huan Zhou

Date: May 3, 2018

1 Introduction

Blockchain is an amazing technology to make every participant having consensus on a worldwide ledger. The Bitcoin, which is the first generation application of blockchain, leverages this ledger to record all the coin transferring history. Then the current balance of a certain account can be calculated through all its related transferring records. These recording types, however, are still too simple to meet requirements of real world transactions. It suffers from the low scalability and poor functionalities, though it is currently the most expensive virtual currency. Therefore, here comes the second generation application of blockchain, Ethereum. It proposes the concept of EVM (Ethereum Virtual Machine), which is a set of byte values to represent a virtual machine state. Then transactions of Ethereum can be interactions from a certain account with the EVM to change its state. This entire process of the state transmission and transaction is recorded in blockchain to make it verifiable and immutable. Hence, user Alice in Ethereum has the ability to define that when some certain condition is met, then the money is transferred to user Bob. This condition is public and no one has the ability to affect the transferring process as long as the condition is met. This is the so called smart contract.
On the other hand, SLA (Service Level Agreement) has been proposed for many years in Cloud computing area. This is an agreement between Cloud client and provider to ensure that some certain service quality can be met, otherwise the client should get compensation from the provider. For instance, the IaaS Cloud provider usually claims that the availability of its data center is more than 95%. If it is not met, they will pay back some credits to the client as compensation. However, this agreement is hard to be enforced in practice. There are following gaps to hinder the traditional SLA to be really adopted in industry.

- **Manual verification:** It lacks of an automatic mechanism to enforce the agreement. For example, the automatic compensation when violation happening. Current process involves a lot of manual work by emails.
- **Rights fairness:** The role of provider has much more rights in this agreement. Because it has the right to verify the violation and decides whether to compensate the client.
- **Proof of violation:** This is an agreement only between client and provider. It is hard for the client to prove that the violation really happens and convince the provider.

In this work, I implement a smart contract [1] based on Ethereum to ensure the client must be compensated when there is violation. This process is automated and immutable. It empowers the right fairness of these two roles in the agreement, especially for the client. Moreover, I propose a model based on witnesses to prove the event of violation. I try to leverage the game theory to demonstrate that the role of witness has to behave honestly, in order to gain the maximum revenue for itself.

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

2 Witness Model for Service Violation

Leveraging Ethereum based smart contract is useful to automate the entire process and ensure the right of client to obtain its compensation. It mostly settles the first two gaps mentioned in Section 1. However, it still remains a difficult problem to mitigate the third gap, which is how to prove the violation truly happens and convince both the client and provider.
Generally speaking, this is also the current challenge for blockchain. That is how to record the random event happening outside the chain and got consensus on the chain. We take an specific example to discuss this.

2.1 Problem scenario

In order to demonstrate this problem, we assume a specific scenario to model this process. In this scenario, the Cloud client requests a VM from the Cloud provider for 10 minutes. Cloud provider provisions a VM and gives the public IP to the client. Meanwhile, the provider promises this VM is always available during the entire service time, 10 minutes. Otherwise, the client automatically gets compensation. But this is the problem that who will monitor the public IP of the VM is accessible during the service time and finally trigger the violation event.
Currently, there is a solution in Ethereum called oracle. It is a trust third party to help the smart contract measure the real world random event and afford the data to the blockchain. This centralized data source solution somehow deviates from the original idea of decentralization in blockchain. It is still hard to convince the participants in a smart contract that the oracle is trustworthy. Moreover, this centralized service may have the problem of single point failure. For example, there might be some network issues of the oracle itself unable to detect the VM and determine there is a violation. However, for most users on the Internet including the client, they still can access the VM. This leads to a misjudgment on the provider, who actually still provides normal services.

2.2 Witness model design

In order to settle these trust and single-point failure issues, we bring in another role, witness, in this scenario. The motivation of this role participating this contract is to gain their own revenue through offering monitoring service. In addition, there are not only one witness, and we design the way how they participate and report violation. According to the game theory, we demonstrate that they have to behave honestly to obtain maximum revenue.
In our model, the witness has the option to decide whether to report the violation according to the monitoring results during the service time. This determines the final revenue can be obtained. We analyze this through taking the example of two witnesses. The revenue matrix of these two witnesses are shown in Table 1.
In this model, we design that the witness can choose two actions, report the violation or not report. Only when most witnesses report the violation event, it is considered

Table 1: The revenue matrix of the witness

$Witness_1$ \ $Witness_2$	Report	Not Report
Report	+10 \ +10	-1 \ +1
Not Report	+1 \ -1	+1 \ +1

to be a truly happened violation. In this case of two witnesses, only when both of them report the violation event, the violation is proved. However, in order to report the violation, the witness must transfer 1 unit money, e.g. 1 coin, to the smart contract to endorse its monitoring results. If it is the minority to report this event, it might be considered as a fraud, then it cannot get its money back as a penalty. In this case, the revenue of the possible fraud is -1 and others get 1 coin for reward. If most of them report the violation, then there will be 10 coins for reward, others will not get money or pay penalty. All the rewards comes from the prepaid money from client and provider. For example, if there are 5 witness, client and provider each should prepay 25 coins to the smart contract. Because 50 is the maximum total reward possible to pay in this case. The remaining rewards and penalties are divided by the client and provider. Since $Witness_1$ and $Witness_2$ cannot communicate with each other, they cannot make an agreement to report at the same time to gain the maximum revenue, if there is actually no violation. Therefore, the Nash equilibria will be (Not Report, Not Report). However, if there is a violation, this is a signal to make witnesses to report this event and pursue the maximum revenue. Moreover, all the actions adopted by these witnesses are recorded by the blockchain. So it is auditable to seek out the frauds. A detailed example is shown in Section 3.2.

Of course, the actual situation should be more witnesses. In addition, in order to ensure that majority of the witnesses do not have relations with the client and the provider or know each other, another witness picking algorithm should be designed to make sure the picked witnesses are random and consented by both the client and provider.

3 Implementation and Experiments

According to the above model design, we implement a prototype smart contract. It provides the functionalities for all of the three roles, the client, provider and witness. The code can be checked from the Github link above.

3.1 Implementation with Solidity

We leverage the language Solidity to implement this smart contract. In order to manage the state of the contract, we design five states. This is useful to make sure that different roles can have different functionalities in different states. These states are as follows.

- **Fresh:** In this state, the provider can prepay its witness service fee and setup SLA. Then the state turns into "Init".
- **Init:** In this state, the client can choose whether to accept this SLA within a time window. If accept, the state turns into "Active". Otherwise, the provider has the right to cancel this SLA, and the state turns back to "Fresh".
- **Active:** In this state, the witness can report a violation event, if there is. The duration of this state depends on the service time.
- **Violated:** If the majority of the witnesses report the violation event within the service duration, the state turns into this "Violated".
- **Completed:** If there is no violation and expires the service duration, the state should be turned into this "Completed".

Then the client, provider and witness can withdraw different amount of money from the smart contract according to the state of "Violated" or "Completed". It is worth to mention that the addresses of the five witnesses is predefined in this smart contract, since this is only a prototype.

3.2 Experiments on Rinkeby

In order to test the code, we download the Ethereum Wallet, create several accounts for different roles. We use the testnet of Rinkeby to conduct this experiment. We request some ETH coins to initialize the accounts.

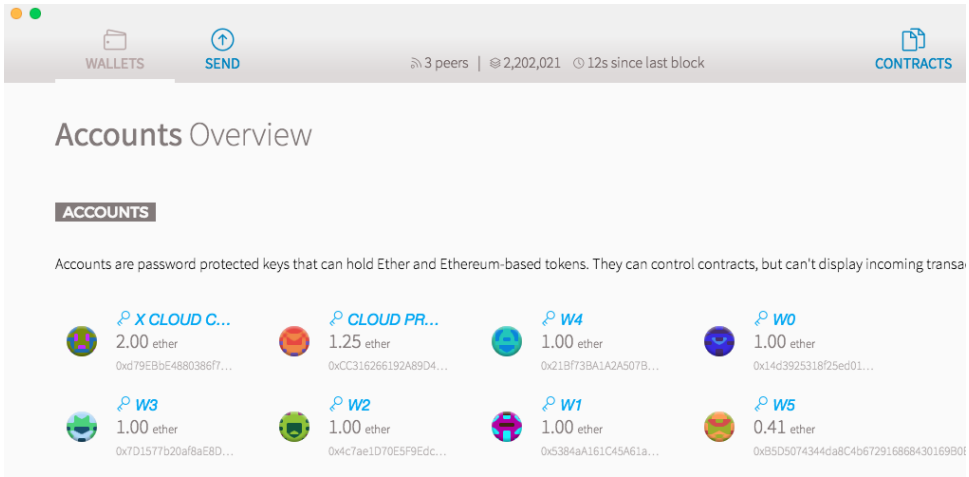


Figure 1: All the accounts overview

The account "X Cloud Client" is for the role of the client. The account "Cloud Provider" is for the role of the provider. The accounts, W_0 to W_5, are for witnesses. Thereinto, W_0 to W_4 are valid witnesses. W_5 is to test the only selected witnesses can join the processes.
First, the Cloud provider deploy this contract and get the contract address.

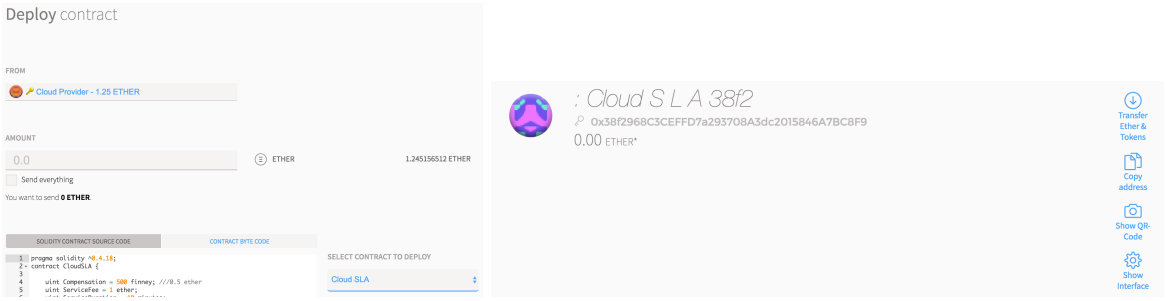
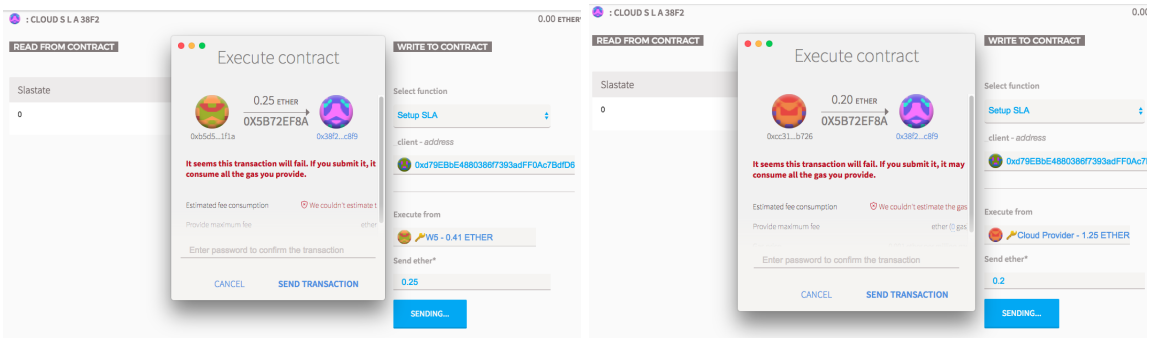


Figure 2: Deploy the contract

In this contract, the service time is 10 minutes. The service fee is 1 ETH. If there is a violation, the compensation fee is 0.5 ETH and the provider can only get 0.5 ETH. 1 unit coin for the witness in Table 1 is 10 finney (1 ETH = 1000 finney). Then the provider can leverage the function of "setupSLA " to pay the prepaid fee for witness service. Then the SLA is set up and waits for the client to accept. It is worth to mention that only the provider can invoke this function and the prepaid money must be correct.



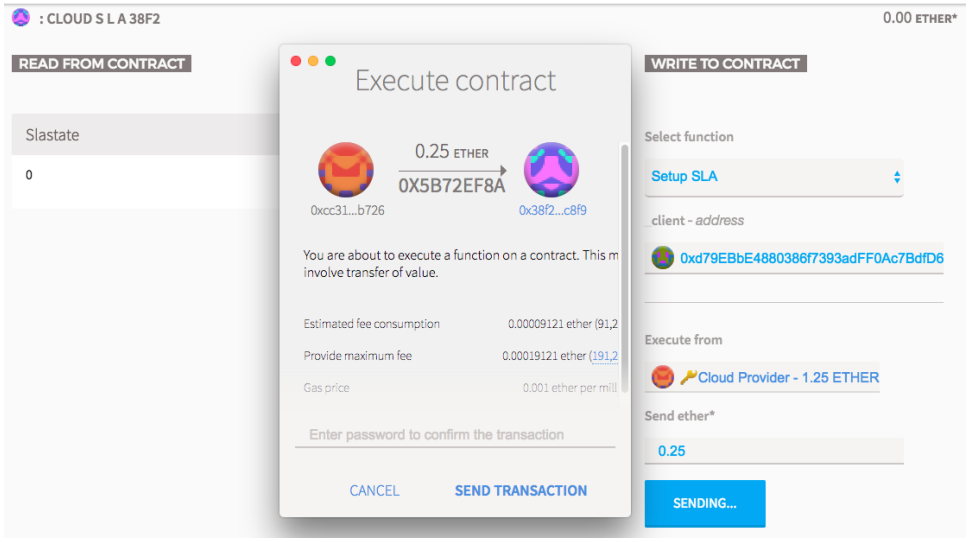


Figure 3: Set up the contract

Then we can capture the event that the state of this SLA contract is modified.

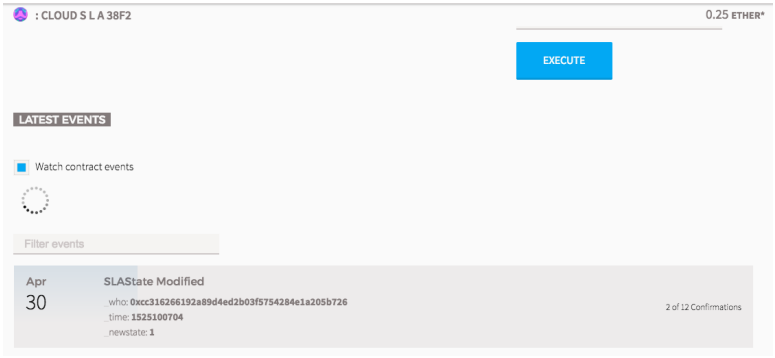


Figure 4: Capture the state event

Now the client has the time window of two minutes. If it has not accepted this in time, the provider has the right to cancel this SLA. The state of the SLA goes back to "Fresh" and provider withdraw back its money.

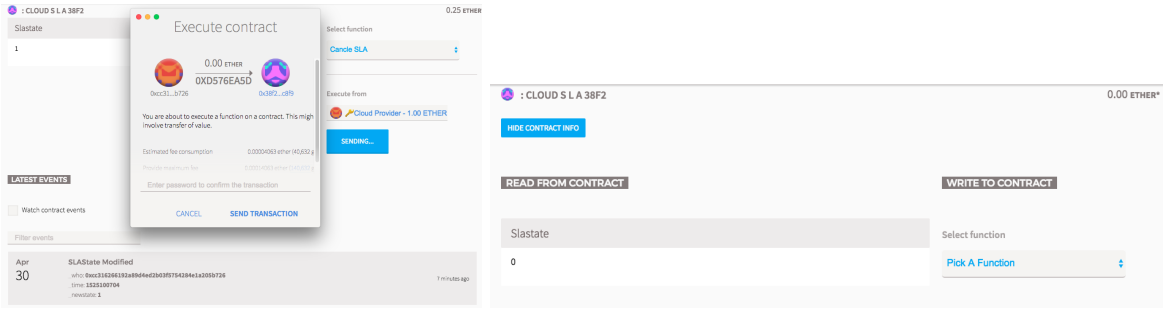


Figure 5: Cancel the SLA

On the other hand, the client can accept this SLA by invoking "acceptSLA" with the prepaid service fee and witness fee, 1.25 ETH (1 ETH + 250 finney). The state of SLA is changed at the same time.

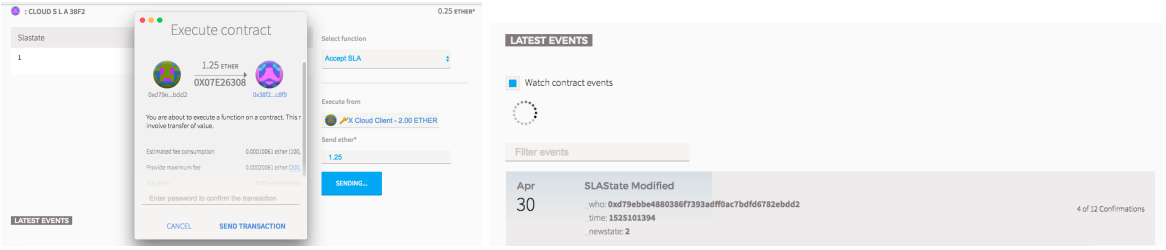


Figure 6: Accept the SLA

Then we assume that there is a violation and W_0 to W_3 report the violation, which is 4 out of 5 report the violation. This will be considered as a real violation event. Therefore, the event is changed to "Violated". Meanwhile, the witness must pay 0.01 ETH (10 finney) to endorse its report.
We also capture the events of reporting the violation.
Therefore, the client should get compensation from this. In this case, the client itself must be the first to withdraw his compensation money. This is through in-

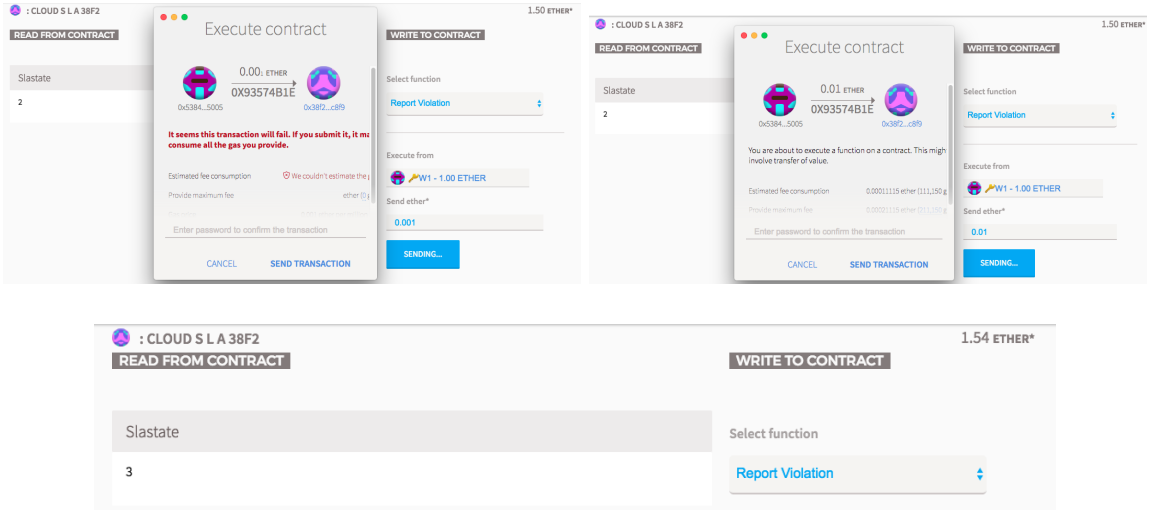


Figure 7: Report the violation

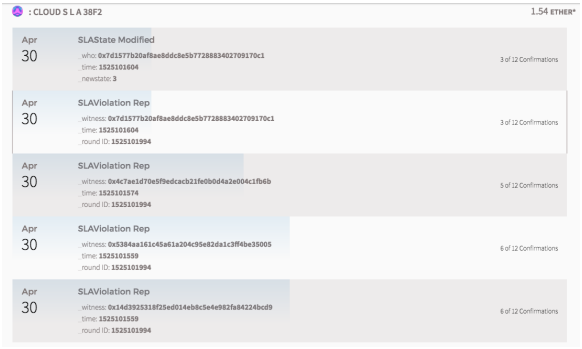


Figure 8: Capture the violation event

voking function “ClientEndVSLAandWithdraw”. Then the state of SLA turns into “Completed”.

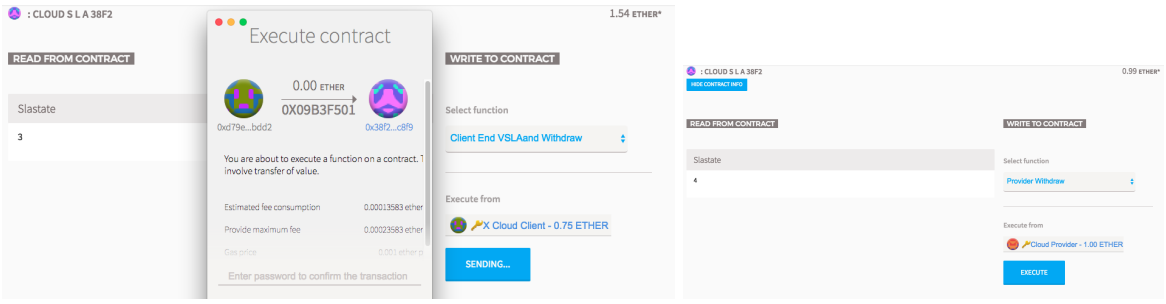


Figure 9: The client ends the SLA and withdraw its compensation

After this, the provider and witness can withdraw money which belongs to them according to the predefined algorithm. Here we also make sure that the only valid witness can withdraw money.
Finally, we show the money in each account to verify the algorithm is correctly implemented.

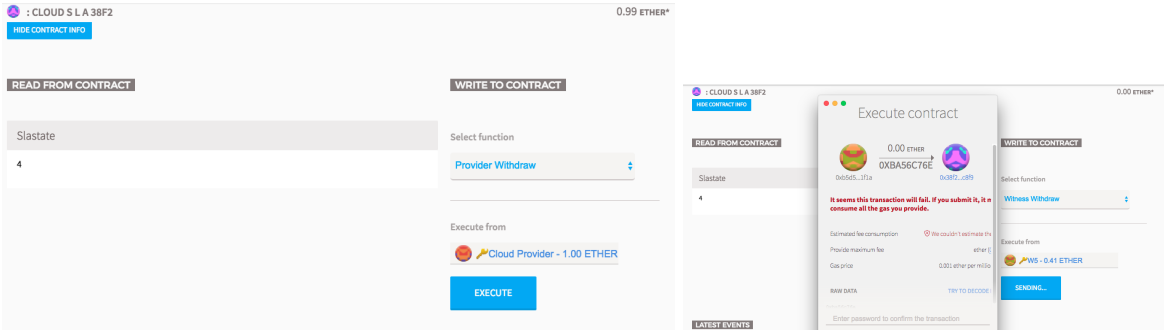


Figure 10: The provider and the witnesses withdraw its compensation

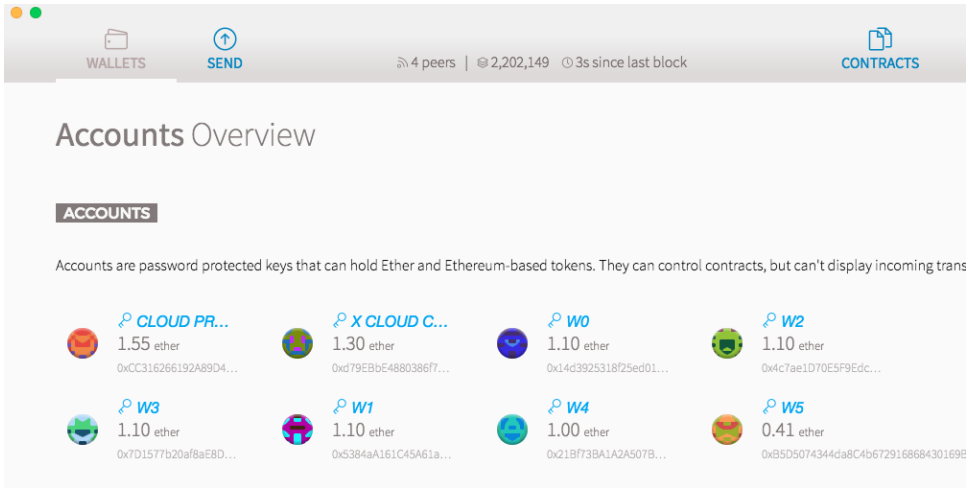


Figure 11: The final account overview

In this case, the 4 out 5 witness got 100 finney rewards because of reporting. The last one does not get reward. The prepaid witness fee, 100 finney, for the last witness is sent back to the provider and client equally. As there is a violation, the client get 0.5 ETH compensation back. And one of the witness does not report the violation. Therefore, the prepaid fee, 100 finney, for this witness will be paid back to the provider and the client. Therefore, the client can get 0.55 ETH back in total. That is the account of the client, “X Cloud Client”, should have 2 - 1.25 + 0.55 = 1.3 ETH. And the provider should have 1 + 0.55 = 1.55 ETH. Figure 11 shows that our implementation results are the same with our model design.