UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

A project report submitted for the award of

Supervisor: Timothy Norman
Examiner: Timothy Norman

Multi Agent Systems on the BlockChain

by Jake Labelle

April 24, 2018

Contents

Chapter 1

Abstract

My project will be a smart contract builder that allows a founder to create a decentralised sports club. This will use the blockchain to provide a trustless environment to collect and allocate money. It will also use multi agent system algorithms to speed up and rationalise the method of choosing representatives and deciding budgets. By combining multi agent systems and blockchain, users will be able to agree to the algorithms before hand and trust that they will be carried out accurately. The project will also look at the cost of running the algorithms and make sure it can scale with the number of representatives and users.
The project will also come with an example web interface that allows users to interact with the block chain without a terminal

Chapter 2

Project Problem and Goals

2.1 Problem

Most university clubs collect registration fees to finance their activities for the year. The current solution for collecting the fees is either with cash or online. Both have the problem that the money is collected by one entity, and cash has the added negative that the payment could be repudiated. Then at the annual general meeting (AGM) , members vote for representatives and for managers. The problem with this is that the AGM is often held before the start of semester one, this means that many members and first years, who are a large proportion of the club members, are unable to vote.
After representatives and managers are chosen, they decide on several places the budget can be spent and how much money each of those places get. The disadvantage with this is that deciding where the money should go can take long time. Another problem with the negotiation phase is that quiet individuals may be bullied into making concessions by someone more confident. Lastly the final budget may not be known by normal members, making it unaccountable and opaque.

2.2 Goals

For the collection of the fees, I want my solution to have these characteristics. First off, all registrations are non repudiable. This means that there is a method for confirming the user has paid the fee. Also we can check the authenticity of a registration. This means that we a user can identify himself as the one who paid

the registration. Another feature is transparency, everyone should be able to see how much money has been collected from fees. Lastly until the budget has been decided no one should be able to access the money.
For the voting, the features are as follows. First off all members who have paid their registration fees should have equal voting power and the ability to apply as a representative. Another feature is the ability for a user to submit a full ranking of representative to allow him a larger ability to customize his vote. We also want to give representatives weights, to take account of the fact that some of them will receive more votes than their counterparts and therefore should have a larger mandate. Lastly the system should discourage tactical voting, as we want people to submit their real votes.
For the negotiation and distribution of the budget, the beneficial characteristics are thus. Representatives can submit sources of expenditures and what their preferred budget is. During this pre negotiation phase all messages sent to each other is recorded and non repudiable, this should increase accountability. Then the system will run a automated negotiation algorithm, which solves the problem of representatives personality affecting the outcome. The negotiation algorithm should be deterministic and therefore others users can check the integrity of the budget. Again the algorithm should be difficult to manipulate by submitting a fake preferred budget. Lastly for the distribution of the budget, after the negotiation the money should be sent automatically to the managers of the expenditures.

Chapter 3

Background Literature

My backgroud reading could be broadly separated into two fields; multi agent systems and smart contract design.

3.1 Multi Agent Systems

3.1.1 An Introduction to MultiAgent Systems Wooldridge (2009)

From this book, there was as description of various voting systems and the desirable properties. First off there was a distinction between social welfare and social choice functions. They both took in a list of preferences from agents, however in social welfare it outputs a list of preferences, whereas social choice outputted a single option. In my system, I will require a social welfare function, as I need to select a number of representatives. For my program, voting preferences are important so that the representatives are representative by of all the voters wishes. Second order Copeland rating satisfies this. This is where each of the options has a pairwise election with all other options. Its then adds the sum of all defeated opponents to see what option is the best. This algorithm satisfies the Condorcet winner condition where w should be ranked first if it would defeat each other option in a pairwise election. This algorithm is Pareto optimal which means that if every one votes $w1 > w2$ then the output should be $w1 > w2$. Lastly, it also mentions the Gibbard-Satterwaite theorem which states that all voting systems are vulnerable to strategic manipulation. For my system, this is made worse by

that fact that all transactions are transparent on a blockchain, making the manipulation even easier. However, second order Copeland rating is NP-complete to manipulate which make it difficult to manipulate.
Also from the book, there was a description on how to run a negotiation algorithm. The negotiation set will be an array of sinks to values, where the values adds up to the collected money. The negotiation algorithm is where Agent 1 makes a proposal. If it is rejected then Agent 2 makes a proposal, then one etc. However if no deal is reached after a certain number of rounds, then the players receive a conflict deal. In the book, two assumptions are made, agents seek to maximise utility and disagreement is the worst outcome. However in my system disagreement is not always the worst outcome. Another important point is that time is valuable and that agents are impatient. This means a deal w at round t1 is better than deal w at round t2 if $w1 < w2$. The algorithm should create a Nash equilibrium, this means that no agent can increase his utility by changing actions.
Lastly, reading continued with the coalition section of the book. My system would be a weighted voting game. This is were each agent has a weight, we have a quota and a coalition C is winning if the sum of the weights exceeds the quota. A coalition is said to be stable if every agent can not get a higher utility by defecting.

3.1.2 Computational Aspects of Cooperative Game Theory Chalkiadakis (2012)

From this book, there was a description of coalition formation by self interested, rational agents. In particular, dynamic coalition forming where coalitions can form and fall apart. One approach is a Markov process, where agents explore suboptimal coalition formation actions and at each stage there is a small probability that they try to move to a new coalition. A player switches coalitions if her expected utility in the best available coalition exceeds her current utility. The player also demands as much as they can from the coalition.

3.2 Smart Contract Design

3.2.1 Pet-shop tutorial Truffle (2017)

I also did research on writing smart contracts, and found the Truffle development environment. I then read through the Pet Shop tutorial . It showed how to set

up the environment, create new contracts with functions and variables, and how to migrate and deploy the contracts. Once the contracts are deployed on Truffle's test network, it gives a number of addresses with Ethereum on them in order to test your contracts. These addresses can then interact with the functions on those contracts. Once you create the contract and deploy them, an interface can be made for the contract with web3. On my system, you will be able to register and vote on the interface, so that users do not have to use the command line. In order to use your Ethereum on a website, downloading the extension Metamask allows you to set your default account, and when you click register on the interface, it will send funds from that account.

3.2.2 Stateless Smart Contracts Childs-Maidment (2016)

In this article, Childs-Maidment describes how he was able to reduce the gas cost of his smart contract. The majority of the gas consumption on smart contracts come from storing data and to reduce the amount of data stored he used IPFS and Stateless Smart Contracts. IPFS take your key-value pair and store it as a hash, and then when access is needed your key-value pair you send the hash to IPFS, and it will return the key-value pair. IPFS is also decentralised so it does not compromise the trust less nature of smart contracts. Stateless Smart contracts have two parts, Dumb functions and filters. A dumb function will take the parameters required but will do nothing. Then when you want to access the data on your interface, since all transactions are public, using InputDataDecoder, find what parameters where passed to the contracts functions.
However, there are trade-offs with this approach. The first being tampering; the person who controls the interface could lie about what was sent. Although users could validate the transactions themselves which reduces the problem. It could also be a positive as anyone can build a filter which could add extra data to improve user experience e.g media, metadata. Another problem is that smart contracts do not have access to all the transactions and will not be able to see your data. Lastly if your contract has no events, it must process all transactions to find your data, but this can be mitigated by firing off random empty events. In my system I will use a mix of standard and stateless functions. The voting and registering will be standard, but for functions such as messaging, gas can be saved by making it stateless.

3.2.3 Ethereum Alarm Clock Merriam (2017)

This was a smart contract Merriam had put on the block chain that allowed you to pay someone to run functions after a certain number of blocks. This would be useful as I need time limits on voting, submitting a budget, term limits and registration and I need to run a function periodically to check that the time limits had not been reached yet.

3.2.4 The ultimate guide to audit a Smart Contract + Most dangerous attacks in Solidity Grincalaitis (2017)

This article described the best way to secure your smart contract from hackers. One common attack is Reentrancy. This is where users can interrupt the normal running of the function. Therefore in my system before any actions occur I will set the conditions to false. Another attack is under and overflows, for uint256 any number above 2**256 will be set to 0 and 0 -1 will be 2**256. Another attack is replay attack. This is where Ethereruem has a hard fork and you use the hard forked Ethereum to send the original Ethereum. Lastly, any function which is not changing the state of the application should be labelled as constant, to save gas.

Chapter 4

Final Design

4.1 How Registration Fee Collected

One of my goals was that the owner of the system does not have any extra control over the money and the data. I will compare the use of a Clubs and Charites Account (Santander) with using a Smart Contract on the Ethereum Blockchain.
Fee collection table here
The benefits of the Clubs and Charites Account is that it costs 0 to run and set up, whereas it can be expensive to store data on the Blockchain. The problem is that only 3 members are required to access the account. This is made worse as the signatories would have to change annually which would require a lot of time and effort. Also if one of the signatories is busy, then the club would be unable to access the funds. Compared to the smart contract, which would do this automatically. Another problem is that although the registration is legally non repudiable, a member would have to take the club to small claims to enforce it. Lastly the account is viewable by the signatories, therefore a regular member can not confirm how much money has been gathered and where it has spent. However a smart contract all members can see how much money has been gathered and where it has been spent. From this I decided that the best design was a smart contract, however care must be taken to keep the Gas (how many instruction used) used down. Also to monitor the Gas Price (cost miners are charging to run a instruction) which this year to date has fallen by 70%, and the price of Ethereum in GBP. Also Another problem to consider is that if a user loses his private key to his Ethereum account it is unrecoverable and you will lose the ability to prove you registered.

4.2 How Registered Users Are Stored

A feature of blockchain is all data stored on it is publicly visible. Therefore, thought must be put into how much data is stored about each user. At minimum we must store the registered users public address. However storing only the users address will make it so communication and accountability is hindered, as a public address is long and non memorable, and will make it hard to see who said and did what. I decided that storing the address and name was the best compromise. In the rare situation two users had the same name and there was argument on who did what, it would not be too much effort for one of them to digitally sign using their private address.

4.3 Voting Algorithm

The voting algorithm to choose representatives is a social welfare function which takes a list of ordered preferences and outputs a ordered preference. I compared a few algorithms to see which best bit my problem.
Voting algo table

4.4 How to start and end Voting

Voting should start a term length after the last voting had ended and the voting should end after voting length after the voting had started. Both term length and voting length are set when the club contract is made. However the smart contract is not constantly running, and therefore needs to have a function called. I have two options for calling the functions. First I could use Ethereum Alarm Clock. This allows my smart contract to pay someone to run the function at a certain block time. The other option is waiting for one of the users to run the start and end vote functions. The first option has the benefit that we can assume that the voting period will end at a more precise time. The latter has the benefit that it is cheaper.
I decided to go for the latter, as in my case a few extra minutes or hours of voting, should not be a significant problem. And if it is, then users could send the function as quickly as possible. If the club has a small number of users the first option might be better as there is a greater chance all the users forget to end the voting period.

4.5 How many Representatives

An important design decision, is how many representatives should be elected. The trade off is between the system being more representative and the system being cheaper to run. Because the voting algorithm compares each candidate in a pair wise election, the gas cost is exponential to the number of representatives. In my system the number of representatives will be set when the contract is created. Also due to the fact in can have a large impact on the cost of the contract, if users wish to change it, a super majority should be acquired.

4.6 How to choose their Weight

After the voting algorithm is complete it will output a list of ranked representatives. From this point the system will need to determine how much weight they have in the budget negotiations. One example system could be weighting the 1st ranked representative, number of representatives, the 2nd, number of representatives 1 and so on. Another system could be weighting them all one. These are common methods and found in numerous other systems and therefore my system should make it easy to choose these two methods. However in order to make the system more flexible, my system should allow the creator to also create their own function.

4.7 How many Sinks and who can create them

Another decision is how many sinks (budget expenditures), should be allowed to be created. Again the system will allow you to choose the number at contract creation. The benefit of more sinks is that the money will more distributed more and each manager will have more time to focus on a sink with less responsibilities. The disadvantage is that communicated between the managers may be poor, and double spending could occur. Another problem is that due to economies of scale, money concentrated in fewer hands may be able to afford more items . lastly it will raise the gas price of the budget negotiation. However since it only affects the cost linearly, the system should only require a plurality to change the number of sinks.
Also the system must have a system for submitting sinks. I have identified 3 ways

to do this. First, all users have the ability, second all representatives have the ability and last, each sink is proposed and then voted on. The first option has the benefit that the system is more interactive for more users. However it is prone to abuse, as being a user only requires paying the fee, and a malicious person could fill up the sinks with bogus sinks. The second option is less like to be abused, as representatives are likely to have a greater attachment to the club. The last option should prevent all abuse, however it is time consuming to have to vote on each sink, and could end up costing a lot of money to run each vote. Therefore I have decided on my second option.

4.8 When can Budget submit end

4.9 Budget Negotiation Algorithm

The first and easiest idea for deciding the budget was to take each representatives budget, sum each one and divide by number of representatives. This would be cheap to run and sounds fair. The problem occurs with that all budgets submitted are publicly viewable, and with this algorithm a representative could easily manipulate his budget to achieve the final budget he wanted. A better algorithm would be for each representative to try and form coalitions with his peers, until the coalition reached a certain quota. This solves the problem of a representative submitting a extreme budget to manipulate the final budget, as the other representatives would just form a coalition without him.
Next I had to decide how coalitions would be formed. The first step would be to initialise each representative with his own coalition with his preferred budget. From this point I had to decide whether representatives would take turns joining coalitions or coalitions would take turns inviting representatives. The negatives of coalitions inviting representatives is that larger coalitions would have more chance to have a representative stolen by another coalition and therefore take longer and cost more to calculate. The advantage is that there would be more negotiations and that should result in a more balanced budget. One problem with both of these solutions is that it could lead to a infinite loop and never end, however the joining coalitions method resulted in a lot less of them. Therefore I will use that method. Representatives will take turns, negotiating with each coalition, they will then compare their utility by joining their coalition and choose the coalition which

offers the best utility, or stay in the current coalition if its better utility then all offers. With the algorithm as follows.

```
Agent[] Agents;
Sink[] Sinks;
Coalitions[] Coalitions;
init Coalitions;
round = 0;
while(winningCoalition = false) {
        player = Agents.get(round % Agents.size());
        //its advantageous to be in a bigger coalition
        currentUtility = utility(player,player.coalition.budget,player.coalition.size);
        currentCoalition = myCoalition;
        for(Coalition c: Coaltions) {
                proposal = negotiate(player,c);
                if(currentUtiltiy < proposal)
                        currentCoalition = c;
                        currentUtiltiy = proposal;
                }
        }
        join current coalition, leave old one
        round +=1
}
```

4.9.1 Calculating Utility

The next step was to create a function which took a representatives budget , their current coalitions budget, and the current size of the coalition and returned the representatives current utility. To make the utility easy to understand, the range of the function will be from 0 to 100. 100 utility would be when the coalition size is the quota and the coalition budget is the representatives preferred budget. The function I decided on was as follows, with a being the representative and b being the coalition.

```
shared = 0;
for(i = 0; i < a.budget.length; i ++)
{
       if(a.budget[i] > b.budget[i])
     {
        shared += b.budget[i];
       }
       else{
        shared += a.budget[i];
       }
}
utility = shared /  quota  + 1 -  b.totalWeight
```

Although representatives joining coalitions reduces the number of infinite loops, any of them are unacceptable in my algorithm. To solve this I came up with two

solutions. First, representatives could be impatient, this means that a budget in round n is better than a budget in round m if n ¡ m. Another method is to make the coalition size less relevant is earlier turns and make it more important in later turns. The first method allows representatives to look forward to later rounds in order to make an more informed decision. However to work out the single negotiation would be factorial, and therefore unusable with a medium number of representatives wheras the second method is a constant effect on the number of instructions and therfore scales much better. For this reason I will use the second method. With the new utility function as follows.

```
shared = 0;
for(i = 0; i < a.budget.length; i ++)
{
       if(a.budget[i] > b.budget[i])
      {
         shared += b.budget[i];
       }
       else{
         shared += a.budget[i];
       }
}
utility = ((coalitionSizeFactor * shared) / (quota + 1 - b.totalWeight) + ((1-coalitionSizeFacto
```

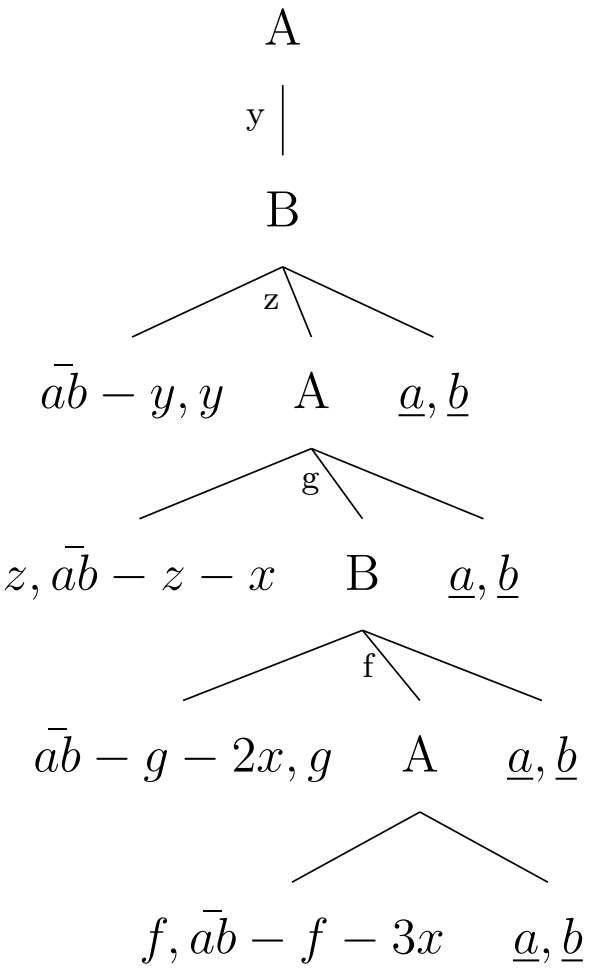
I also need to set the coalition size factor when the algorithm is started, and need to increase it each round.

4.9.2 Alternating Offers Game

The next decision was how representatives would negotiate with coalitions. I used the model of the alternating offers game with impatient agents. I will therefore use back propagation to work out what the optimal starting offer is for both representative a and coalition b . First I work out $\bar{a}, \bar{b}, \underline{a}, \underline{b}$, these are in order, the max utlity a can get from joining b, the least utlity a can get from joining b, the current utility that a has in its current coalition, and the current utility of b.

$$\bar{a} = utiltiy(a.budget, a.budget, b.totalWeight + a.weight)$$
$$\bar{b} = utility(a.budget, b.budget, b.totalWeight + a.weight)$$
$$\underline{a} = utility(a.budget, a.currentCoalition.budget, a.currentCoaltion.totalWeight)$$
$$\underline{b} = utility(b.budget, b.budget, b.totalWeight)$$
$$\bar{ab} = \bar{a} + \bar{b}$$

The next step is to now check if $\bar{ab} < \underline{a} + \underline{b}$, if so then we can stop the negotiation as there is no way for both the representative and the coalition to benefit. We also need the variables x and N, which are the cost to run one turn one offer, and the number of turns that are run. With this we can generate this tree. We will set N = 4 in order to make it easier to display.



From this we can find n, that is the turn in which one of the agents can offer the other agent a score higher than the conflict deal for the both of them, $\underline{a}, \underline{b}$. We can rearrange $\bar{ab} - (n-1)x - \underline{a} - \underline{b}$ to $(\bar{ab} - \underline{a} - \underline{b} - 2)/x = n$. If n is greater than N, then we can set n = N, as it can not go over that many turns.
The next step is to calculate the free utility, that is how much extra utility can the agent take for himself on the turn that the offer better than the conflict deal can occur. This is $freeUtility = \bar{a,b} - (n-1)x - \underline{a} - \underline{b} - 2$. Now there are two cases, if n is even then b takes the free utility, and if its off then a takes it. Then for the remaining turns the offeror takes x -1 extra utility and the offeree gets 1 extra utility. Therefore I get this function for the optimal utility for each agent.

```
if(n \% 2 ==0)
{
```

```
     aUtility = \underline{b} + 1 + freeUtility + (((n/ 2)-1) * x) + 1;
     bUtility = \bar{a} + \bar{b} -  aUtility;
}
else
{
      bUtility = \underline{a}  + 1 + freeUtility +  ((((n + 1)/ 2)-1) * x);
      aUtility = \bar{a} + \bar{b} -  bUtility;
 }
```

The last flaw I had to fix was that the algorithm could give a utility higher than $\bar{a}$, which should be impossible. Therfore if either aUtility or bUtility is over $\bar{a}$, reduce it to $\bar{a}$ and set the other one to $\bar{b}$.

Chapter 5

Implementation

5.1 Setting up Development Environment

In order to create and migrate the smart contract onto the blockchain I used a tool called Truffle. This had boilerplate code which had a .sol contract file and a initial_migration.js and a deploy_contracts.js. After writing code in the .sol file and making sure to deploy in in the deploy_contracts.js, you could then run truffle compile to create .json build files which contains fields such as the abi. This could then be deployed to the real blockchain, however while in development, Truffle has a tool develop, which allows you to write to deploy the smart contract to a local network. It also provides several addresses preloaded with Ethereum, this allows me to test my program without having to pay any money.

5.2 Club Contract Inheritance

I decided to split my Club smart contract into three, RegistrationClub, VotingClub and BudgetClub. With BudgetClub inheriting from VotingClub and VotingClub inheriting from RegistrationClub. The first reason this was done was to increase readability. Another reason is that I want to make it easier for developers to adapt my smart contract to their situation. E.g A completely different way of choosing representatives, would mean that only VotingClub needs to be changed.
I also created a library called MyLib which will contain all my structs.

5.3 Registration Club

This contract needs to take a registration cost when its created, and then allow users to pay that much Ethereum to register. It then needs to store that user in a array as a Struct containing the users name and address. I also require getters to return the number of registered uses, a certain registered user and the balance in the smart contract. This was done with the following code

```
contract RegistrationClub {

  MyLib.User[] registeredUser;
  uint public registrationCost;

  function RegistrationClub(uint cost) public {
    registrationCost = cost;

  }

  function register(string name) public payable {
    if (msg.value > registrationCost && (!isRegistered(msg.sender,registeredUser)) ) {
      MyLib.User memory u;
      u.myAddress = msg.sender;
      u.name = name;
      registeredUser.push(u);
    }
  }


  function isRegistered(address sample, MyLib.User[] userList) internal pure returns (bool)
  {
    for (uint i = 0; i < userList.length; i++)
    {
      if(userList[i].myAddress == sample)
      {
        return true;
      }
    }
    return false;
  }
  function listRegisteredUsers(uint number) public constant returns (string,address) {
  return (registeredUser[number].name, registeredUser[number].myAddress);
  }

  function getRegisteredUsersLength() public constant returns (uint)
  {
     return registeredUser.length;
   }
   function getContractValue() public constant returns (uint)
   {
      return this.balance;
   }
}
```

I also needed to add the functionality to allow users to change the registration cost if a super-majority of users wish it. First I added a global mapping which takes a registered user index and returns their current suggestion that user has for the new registration cost. When a new user registered I will set their suggestion to the current registration cost. After that I needed the ability to submit a new registration cost suggestion, and another to check all the submissions and check if a super majority of the same submissions has been reached. This was done with the code as follows.

```
function changeRegistrationCostSuggestion(uint suggestion) public {
    for (uint i = 0; i < registeredUser.length; i++)
    {
      if(registeredUser[i].myAddress == msg.sender)
      {
        registrationCostSuggestions[i] = suggestion;
      }
    }
  }

  function changeRegistrationCost() public {
    uint suggestion = checkRegistrationSuggestions();
    if(suggestion != registrationCost)
    {
      registrationCost = suggestion;
    }
  }

  function checkRegistrationSuggestions() constant public returns (uint) {
      uint superMajority = registeredUser.length / 4 * 3;
      for (uint i = 0; i < registeredUser.length; i++)
      {
        uint count = 0;
        uint iSuggestion = registrationCostSuggestions[i];
        for (uint p = 0; p < registeredUser.length; p++)
        {
          if(iSuggestion == registrationCostSuggestions[p])
          {
            count ++;
          }
        }
        if (count > superMajority)
        {
          return iSuggestion;
        }
      }
      return registrationCost;
  }
```

An important part of the code is that checkRegistrationSuggestions() is constant and public. This is important as running changeRegistrationCost() costs gas even if no changes are made to the registration cost. Therefore a user only wants to run changeRegistrationCost() when he is certain that it will change the registration

cost. A user can do this by running checkRegistrationSuggestions() for free and seeing if the return is a value different than the current registration cost.

5.4 Voting Club

5.5 Budget Club

5.6 Club Contract Website

Chapter 6

Testing

Chapter 7

Conclusions

It works.

Chapter 8

Future Work

Bibliography

Georgios Chalkiadakis. Computational aspects of cooperative game theory, 2012.

James Childs-Maidment. Stateless smart contracts, 2016.

Merunas Grincalaitis. The ultimate guide to audit a smart contract + most dangerous attacks in solidity, 2017.

Piper Merriam. Etherum alarm clock, 2017.

Truffle. Pet shop tutorial, 2017.

Michael Wooldridge. An introduction to multiagent systems, 2009.