

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 27, 2018

Contents

1	Abstract	1
2	Project Problem and Goals	2
2.1	Problem	2
2.2	Goals	3
3	Background Literature	4
3.1	Multi Agent Systems	4
3.1.1	An Introduction to MultiAgent Systems Wooldridge (2009)	4
3.1.2	Computational Aspects of Cooperative Game Theory Chalkiadakis (2012)	5
3.2	Smart Contract Design	5
3.2.1	Pet-shop tutorial Truffle (2017)	5
3.2.2	Stateless Smart Contracts Childs-Maidment (2016)	6
3.2.3	Ethereum Alarm Clock Merriam (2017)	7
3.2.4	The ultimate guide to audit a Smart Contract + Most dangerous attacks in Solidity Grincalaitis (2017)	7
4	Final Design	8
4.1	How Registration Fee Collected	8
4.2	How Registered Users Are Stored	9
4.3	Voting Algorithm	9
4.4	How to Start and End Voting	10
4.5	How many Representatives	10
4.6	How to Choose their Weight	10
4.7	How Many Sinks and Who Can Create Them	11
4.8	When can Budget Submit end	11
4.9	Budget Negotiation Algorithm	12
4.9.1	Calculating Utility	13
4.9.2	Alternating Offers Game	13
5	Implementation	16
5.1	Setting up Development Environment	16
5.2	Club Contract Inheritance	16
5.3	Registration Club	17

5.4	Voting Club	19
5.5	Budget Club	22
5.6	Club Contract Website	28
6	Testing	29
7	Conclusions	30
8	Future Work	31
	Bibliography	32

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 Ethereum 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.

Ethereum is a crypto currency which allows users to upload smart contracts to the network. Other users can then send transactions to the smart contract. These transactions are validated by miners in return for a miners fee, a certain amount of Ethereum. The miners then run the transaction and update the storage of the smart contract.

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. The problem for both systems is that the money is collected by one entity, and cash has the added negative that the payment could be repudiated. At the annual general meeting (AGM) , members vote for representatives and for managers. A problem arises when the AGM is held before the start of semester one, and this means that many members and first years students, who are a large proportion of the club members, are unable to vote.

After representatives and managers are chosen, they will decide on several items where budget can be spent and how the money is allocated to those items. The disadvantage with this, is that deciding where the money should go, can take a long time. Another problem with the negotiation phase is that quiet individuals may be bullied into making concessions by someone more confident. Another problem is that currently a lot of systems use the budget negotiation algorithm of take all the representatives budget and divide by the number of representatives. This is very easy to manipulate by overexaggerating one expenditure, in order to make that expenditure larger in the final budget. Lastly, the final budget may not be known by the entirety of the club, making it unaccountable and opaque.

Another problem is that current systems run on private servers and databases, and therefore users can not trust that the owners of the systems are not manipulating the data. Also the servers and databases require a upfront cost.

2.2 Goals

For the collection of the fees, I want my solution to have certain characteristics. First off, all registrations are non-repudiable. This means that there is a method for confirming that the user has paid the fee. Also, we can check the authenticity of a registration. This means that 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; Firstly, 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 representatives, to allow him a larger ability to customize his vote. We also want to give each representatives a weight, to take into account 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 votes that represent their opinion, and it not swayed by how other people are voting.

For the negotiation and distribution of the budget, the beneficial characteristics are thus; representatives can submit sources of expenditures and their preferred budget. During this pre-negotiation phase, all messages sent to each other are recorded and non-repudiable, and this should increase accountability, because representatives will be held to the claims they made. The system will then run an automated negotiation algorithm, which solves the problem of representatives personality affecting the outcome. The negotiation algorithm should be deterministic and therefore other users can check the integrity of the budget. The algorithm should make it difficult to manipulate the final 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 background 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 $w_1 > w_2$ then the output should be $w_1 > w_2$. 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 t_1 is better than deal w at round t_2 if $w_1 < w_2$. 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 where 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 were 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 is 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.

Policy ID	Clubs and Charites Account	Smart Contract
Signatories To Access Money	3	Cant unless given the money by the budget.
Max Turnover	250,000	Unlimited
Cost to Run and Withdraw	0	Gas Used per Transaction * Gas Price (Currently 3)
Registration is non repudiable	Legally yes, but practically nothing to stop someone from denying registration happened.	Yes
Able to prove youre the one who registered	Yes	Yes
Transparency (all users can see in and outflows)	No	Yes

The benefits of the Clubs and Charites Account is that it costs zero 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 considerable time and effort. Also, if one of the signatories is unavailable, then the club would be unable to access the funds. Compared with the Smart Contract, which this is done automatically. Another problem is that although the registration is legally non repudiable, a member would have to take the club to small claims court to enforce it. Lastly, the account is viewable by the signatories, therefore not all members can not confirm how much money has been gathered and where it has spent. However, with 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 instructions 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 registration.

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 solved this 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 a preset time after the voting has started. Both term length and voting length are set when the club contract is made. However Smart Contract functions can not run for a long time, and so to check if enough time has past, someone will have to call the function after enough time has past. I have two options for calling the time sensitive 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. If it is, then users could send the function as quickly as possible to mitigate the problem. 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 it 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 widely distributed and each manager will have more time to focus on a sink with less responsibilities. The disadvantage is that communication 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 method for submitting sinks. I have identified three 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 for. 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 the second option.

4.8 When can Budget Submit end

This was solved the same method as how voting was ended.

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 that occurs is 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 to 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: Coalitions) {
        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. I wanted the utility to capture that users are happier the more their budget overlaps with their final budget. 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.

$$Utility = (\sum_{n=0}^k \max(a.budget[n].b.budget[n])) / (quota + 1 - a.weight - b.totalWeight)$$

Although representatives joining coalitions reduces the number of infinite loops, any of them are unacceptable in my algorithm. To solve this I propose 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 in 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 a more informed decision. However to work out the single negotiation would scale factorial, and therefore unusable with a medium number of representatives whereas the second method is a constant effect on the number of instructions and therefore scales much better. For this reason, I will use the second method, with the new utility function as follows.

$$Overlap = (\sum_{n=0}^k \max(a.budget[n].b.budget[n]))$$

$$Utility = ((coalitionSizeFactor * Overlap / (quota + 1 - b.totalWeight)) + ((1 - coalitionSizeFactor) * Overlap))$$

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 utility a can get from joining b, the least utility a can get from joining b, the current utility that a has in its current coalition, and the current utility of b.

$$\bar{a} = utility(a.budget, a.budget, b.totalWeight + a.weight)$$

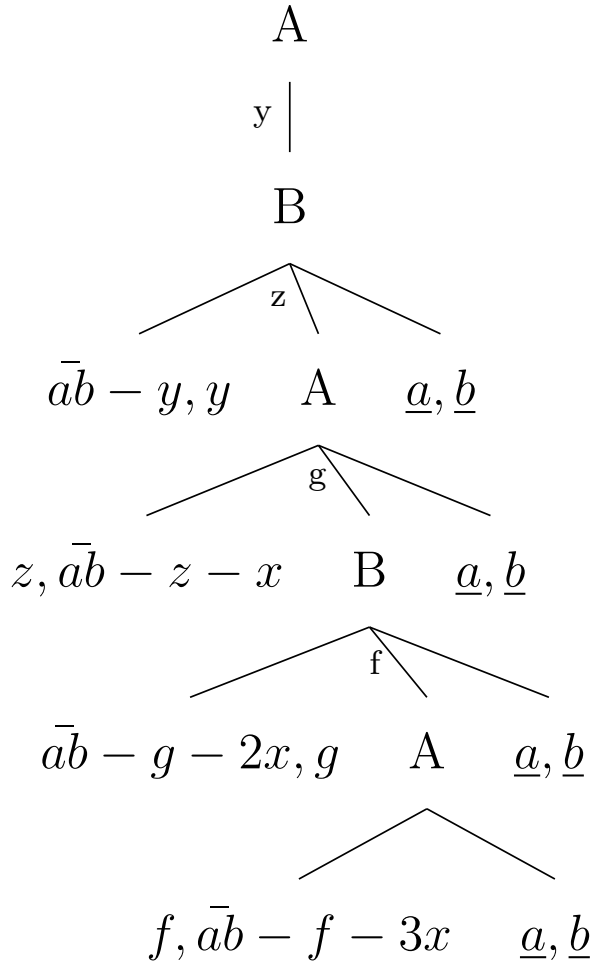
$$\bar{b} = utility(a.budget, b.budget, b.totalWeight + a.weight)$$

$$\underline{a} = utility(a.budget, a.currentCoalition.budget, a.currentCoalition.totalWeight)$$

$$\underline{b} = utility(b.budget, b.budget, b.totalWeight)$$

$$\bar{ab} = \bar{a} + \bar{b}$$

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 of negotiation, and the number of turns that are run. With this information 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 $\bar{a}\bar{b} - (n-1)x - \underline{a} - \underline{b}$ to $(\bar{a}\bar{b} - \underline{a} - \underline{b} - 2)/x = n$. 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 $freeUtility = \bar{a}, \bar{b} - (n-1)x - \underline{a} - \underline{b} - 2$. 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 needed to fix was that the algorithm could give a utility higher than \bar{a} , which should be impossible. Therefore 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

The basic features this needs to implement is the ability to start the voting, each user to submit one list of ranked users, end voting and then rank each user by their second order Copeland score. After ranking their score they will be given a weight according to the algorithm submitted (or with a default one). It will also take three variables, the registration cost, the term length and the number of representatives.

One problem that was spotted was that second order Copeland does pairwise comparison on each agent. This means we could be comparing users who have no votes which could cost a lot of gas. To solve this problem I added another list of users candidates. Any user can become a candidate and users can only vote for candidates. This greatly reduces the average number of users I have doing pairwise comparison on. Also since storage is expensive on smart contract I will store the candidates as a array of registeredUsers indexes instead of the User struct. My apply method is as follows.

The `startVote` function is as follows.

```
function startVote() public {
    if(timeVoteEnded + termLength < block.timestamp && voteStarted == false)
    {
        voteStarted = true;
        timeVoteStarted = block.timestamp;
    }
}
```

The `block.timestamp` is insecure as a malicious actor with a large amount of computing power could create fake blocks with the wrong timestamp. However I decided this was not of huge concern, as it would cost a lot of energy and money and all the actor would be able to do is start and end the voting a few minutes or seconds early or late.

The `Vote` function is as follows.

```
function vote(address[] myVote) public {
    for (uint i = 0; i < registeredUser.length; i++)
    {
        if(registeredUser[i].myAddress == msg.sender)
        {
            if(voteStarted == true && registeredUser[i].voted == false){
```

```

        registeredUser[i].voted = true;
        votes.push(myVote);
    }
}
}

```

The endVote function is as follows.

```

function endVote() public
{
    if(timeVoteStarted + votingTime <= block.timestamp && voteStarted == true)
    {
        voteStarted = false;
        timeVoteEnded = block.timestamp;
        if(candidates.length < numberOfRepresentatives)
        {
            numberOfRepresentatives = candidates.length;
        }

        countVotes();

        candidates.length = 0;
        votes.length = 0;
    }
}

```

The countVotes function is as follows.

```

function countVotes() internal {
    uint i;
    uint p;
    uint[] memory copelandScore = new uint[](candidates.length);
    uint[][] memory defeated = new uint[][](copelandScore.length);

    //init2d array
    for (i = 0; i < copelandScore.length; i++)
    {
        defeated[i] = new uint[](copelandScore.length);
    }

    //pairwise each candidate against each other
    for (i = 0; i < copelandScore.length - 1; i++)
    {
        for (p = 1 + i; p < copelandScore.length; p++)
        {
            uint pairwise = pairwiseComparison(i,p);

            if(pairwise == 0)
            {
                defeated[i][copelandScore[i]] = p;
                copelandScore[i] ++;
            }
        }
    }
}

```

```
        else if (pairwise == 1) {
            defeated[p][copelandScore[p]] = i;
            copelandScore[p] ++;
        }
    }
}
```

In order to save gas I implemented a number of strategies. One method is to only load data from storage once. This is why I used `copelandScore.length` instead of `candidates.length`. As `copelandScore.length` is stored in memory. Memory load costs 3 gas whereas Storage load costs 50. This however lead to another problem, as the solidity language only allows 7 items of data on the stack. To solve this is reused counter variables and split the count vote into separate functions. Another way I saved gas was by moving the initialising the `p` counter as `1 + i`. This made it so my I only had to do $n(n + 1)/2$ pairwise comparisons to calculate all the Copeland scores. This can be done because a defeat for one candidate increases the Copeland score of the other. If this was not done it would have taken n^2 pairwise comparison. Also during the pairwise comparison I reduced gas cost by stopping counting votes when either of the candidate has been rated higher in over half the votes. We can do this because Copeland score does not care about the quality of pairwise victories, but the quantity. This does not reduce the worst case gas cost, however the majority of times it should have significant impact.

Another decision I made was to break the for loop as soon as either candidate `i` or candidate `ps` address has been found. This is because in Copeland score we do not need to know how much more a candidate is ranked higher, only if it is. This again saved gas.

After finding the Copeland score of each user and an array for each candidate of the candidates they defeated. I worked out the second order Copeland of each candidate by summing the Copeland score of their defeated candidates.

One problem that I faced now was that I could not just simply sort the second order Copeland scores. This is because to save gas all my variables stored the candidate list user index instead of the users themselves. This meant if I just sorted the scores, I would lose who had what score. To solve this problem I sorted them, then compared the sorted and unsorted scores to create another array called `ranked`, which stored the list of candidate indexs, ranked by second order Copeland scores. I used quick sort to sort as it has the best average case time complexity $O(n \log n)$, it also has space complexity of $\log n$. Also while testing I realised a error occurred where if multiple candidates had the same second order Copeland

score, they would all be ranked to the first candidate in the list. To solve this I added; `secondOrderCopeland[p] = copelandScore.length * copelandScore.length;` The last function to implement was how to weight the candidates. I decided upon creation of the contract the creator can choose options, with developers being able to add more options. By default I added the two simplest options, each representative getting 1 weight, and each representative get decreasingly less weight. The function is as follows.

```
function weightCandidates(uint[] ranked) public
{
    uint weightingOptionMemory = weightingOption;
    for (uint i = 0; i < numberOfRepresentatives; i++)
    {
        MyLib.Agent memory a;
        a.u = candidates[ranked[i]];
        if(weightingOptionMemory == 0)
        {
            a.weight = 1;
        }
        else if(weightingOptionMemory == 1)
        {
            a.weight = numberOfRepresentatives - i;
        }
        listOfRepresentatives.push(a);
    }
}
```

One other important feature of solidity is that storage variable creation costs 20,000, but resizing the array length to 0 costs 5000. This means that cost of the first run of the system will cost more than the usual amount.

5.5 Budget Club

This smart contract had to allow representatives to submit sinks, and preferred budgets. It then had to run my decide budget algorithm. Finally after a final budget is reached, the contract must send out the money according to the budget. The code for submitting budgets and sinks is as follows.

```
function submitBudget(uint[] budget) public
{
    if(budget.length <= numberOfSinks)
    {
        for (uint i = 0; i < numberOfRepresentatives; i++)
        {
            if(registeredUser[listOfRepresentatives[i].u].myAddress == msg.sender)
            {
                listOfRepresentatives[i].budget = budget;
            }
        }
    }
}
```

```

    }
}
function submitSink(address spender, string name) public
{
    if(listOfSinks.length < numberOfSinks)
    {
        for (uint i = 0; i < numberOfRepresentatives; i++)
        {
            if(registeredUser[listOfRepresentatives[i].u].myAddress == msg.sender && listOfRepresent
            {
                MyLib.Sink memory s;
                s.name = name;
                s.spender = spender;
                listOfSinks.push(s);
                listOfRepresentatives[i].sinkSubmitted += 1;
            }
        }
    }
}

```

Only representatives can do these actions. Also to stop representatives from abusing their ability and filling up the number of sinks with their own addresses, I created a variable `numOfSinksAllowedToSubmit` which is set to `sinks / reps + 1` and tracked how many sinks each representatives had already submitted.

The next code to implement was the end budget submit, this can be done after a certain amount of time after voting had ended. With the code as follows. I checked that the list of representatives was not empty to make sure that users did not run this function after it had already been run once.

```

function endBudgetSubmit() public
{
    if(listOfRepresentatives.length != 0 && timeVoteEnded + 10000 < block.timestamp)
    {
        decideBudget();
        listOfRepresentatives.length = 0;
    }
}

```

Next I had to implement the decide budget algorithm. It is as follows.

```

function decideBudget() public
{
    uint quotaMemory = quota;
    uint numberOfRepresentativesMemory = numberOfRepresentatives;
    uint[][] memory coalitionBudgets = new uint[][](numberOfRepresentativesMemory);
    uint[] memory coalitionSize = new uint[](numberOfRepresentativesMemory);
    uint[] memory playerToCoalition = new uint[](numberOfRepresentativesMemory);
    MyLib.Agent[] memory playerToAgent = new MyLib.Agent[](numberOfRepresentativesMemory);
    uint i;
    //init
    for (i = 0; i < numberOfRepresentativesMemory; i++)
    {

```

```

    playerToAgent[i] = listOfRepresentatives[i];
    coalitionBudgets[i] = playerToAgent[i].budget;
    coalitionSize[i] = 1;
    playerToCoalition[i] = i;
}

uint player = 0;
while(winningCoalition(playerToCoalition,numberOfRepresentativesMemory,playerToAgent) == num
{
    uint[] memory bestOffer = new uint[](2);
    uint bestCoalition = numberOfRepresentativesMemory;
    //get utility in the current coalition
    uint myAfloor = afloor(coalitionBudgets[playerToCoalition[player]],playerToAgent[player].b
    for(i = 0; i < numberOfRepresentativesMemory; i ++)
    {
        //dont check coalitions with no one or the one your in
        if(playerToCoalition[player] != i && coalitionSize[i] != 0)
        {
            uint[] memory nego = offer(coalitionBudgets[i],playerToAgent[player].budget,myAfloor,c
            if(nego[0] > bestOffer[0])
            {
                bestOffer = nego;
                bestCoalition = i;
            }
        }
    }

    if(bestCoalition != numberOfRepresentativesMemory)
    {
        coalitionBudgets[bestCoalition] = negoToBudget(playerToAgent[player].budget, coalitionBu
        coalitionSize[playerToCoalition[player]] -= playerToAgent[player].weight;
        coalitionSize[bestCoalition] += playerToAgent[player].weight;
        playerToCoalition[player] = bestCoalition;
    }

    if(player == numberOfRepresentativesMemory - 1)
    {
        player = 0;
    }
    else{
        player ++;
    }
    coalitionSizeFactor += coalitionSizeFactorIncrease;
}

finalBudget = coalitionBudgets[winningCoalition(playerToCoalition,numberOfRepresentativesMem

}

function offer(uint[] budgetCoal, uint[] budgeta, uint myAfloor, uint coalSize, uint agentWeig
{
    uint quotaMemory = quota;
    uint myAbar = abar(coalSize, agentWeight, quotaMemory);
    uint myBbar = bbar(budgeta, budgetCoal, coalSize, agentWeight, quotaMemory);
    uint myBfloor = bfloor(coalSize, quotaMemory);

```

```

    if(myAfloor + myBfloor < myAbar + myBbar)
    {
        return negotiate(myAbar, myBbar ,myAfloor, myBfloor);
    }
    else{
        return new uint [] (2);
    }
}

function afloor(uint[] budgetCoala, uint[] budgeta, uint coalSizea, uint quotaMemory ) constant
{
    return calculateUtility(budgetCoala,budgeta,coalSizea,quotaMemory);
}

function bfloor(uint coalSize, uint quotaMemory) constant internal returns (uint)
{
    return calculate100Utility(coalSize,quotaMemory);
}

function abar(uint coalSize, uint agentWeight, uint quotaMemory) constant internal returns (uint)
{
    return calculate100Utility(coalSize + agentWeight,quotaMemory);
}

function bbar(uint[] budgetCoal, uint[] budgeta, uint coalSize, uint agentWeight, uint quotaMemory)
{
    return calculateUtility(budgetCoal,budgeta,coalSize + agentWeight,quotaMemory);
}

function winningCoalition(uint[] playerToCoalition, uint numberOfRepresentativesMemory, MyLib.A)
{
    uint[] memory weights = new uint [] (numberOfRepresentativesMemory);
    //add all the weights
    for (uint i = 0; i < numberOfRepresentativesMemory ; i++)
    {
        weights[playerToCoalition[i]] += playerToAgent[i].weight;
    }
    for (i = 0; i < numberOfRepresentativesMemory ; i++)
    {
        if(weights[i] >= quota)
        {
            return i;
        }
    }
    return numberOfRepresentativesMemory;
}

function calculate100Utility(uint coalitionSize, uint quotaMemory) constant internal returns
{
    return (uint) ((coalitionSizeFactor * 100) / (quotaMemory + 1 - coalitionSize) + ((100-coalitionSizeFactor) * 100) / (quotaMemory + 1 - coalitionSize));
}

function calculateUtility(uint[] budget1,uint[] budget2, uint coalitionSize, uint quotaMemory)
{
    uint shared = sharedBudget(budget1,budget2);
    return (uint) ((coalitionSizeFactor * shared) / (quotaMemory + 1 - coalitionSize) + ((100-coalitionSizeFactor) * 100) / (quotaMemory + 1 - coalitionSize));
}

function sharedBudget(uint[] budget1,uint[] budget2) constant internal returns (uint)
{
    uint shared = 0;
    for(uint i = 0; i < budget1.length; i ++)
```

```

    {
        if(budget1[i] > budget2[i])
        {
            shared += budget2[i];
        }
        else{
            shared += budget1[i];
        }
    }
    return shared;
}

function negotiate(uint a_bar,uint b_bar,uint a_floor,uint b_floor)constant public returns (u
{
    uint gasCostMemory = gasCost;
    uint usefulTurn = ((a_bar + b_bar - a_floor - b_floor - 2) / gasCostMemory) + 1;
    uint[] memory proposal = new uint[](2);

    uint numberOfTurnsMemory = numberOfTurns;
    if(usefulTurn > numberOfTurnsMemory)
    {
        usefulTurn = numberOfTurnsMemory;
    }

    uint freeUtility = a_bar + b_bar - ((usefulTurn - 1) * gasCostMemory) - a_floor - b_floor -
    if(usefulTurn % 2 == 0)
    {
        proposal[1] = b_floor + 1 + freeUtility + (((usefulTurn/ 2)-1) * gasCostMemory) + 1;
        proposal[0] = a_bar + b_bar - proposal[1];
    }
    else
    {
        proposal[0] = a_floor + 1 + freeUtility + (((usefulTurn + 1)/ 2)-1) * gasCostMemory);
        proposal[1] = a_bar + b_bar - proposal[0];
    }
    //cant go over the best amount
    if(proposal[0] > a_bar)
    {
        proposal[1] += proposal[0] - a_bar;
        proposal[0] = a_bar;
    }
    else if(proposal[1] > a_bar)
    {
        proposal[0] += proposal[1] - a_bar;
        proposal[1] = a_bar;
    }

    return proposal;
}

function negoToBudget(uint[] budget1, uint [] budget2, uint[] nego, uint coalitionSize) consta
{
    uint[] memory newBudget = new uint[](budget1.length);

    uint sharedab = sharedBudget(budget1,budget2);

```

```

uint quotaMemory = quota;
uint individuala = (uint) (nego[0] / ((coalitionSizeFactor / (quotaMemory + 1 - coalitionSiz
uint individualb = (uint) (nego[1] / ((coalitionSizeFactor / (quotaMemory + 1 - coalitionSiz

uint suma = 0;
uint sumb = 0;

for(uint i = 0; i < budget1.length; i ++)
{
    if(budget1[i] > budget2[i])
    {
        suma += budget1[i];
    }
    else{
        sumb += budget2[i];
    }
}

uint total = 0;
for(i = 0; i < budget1.length; i ++)
{
    if(budget1[i] > budget2[i])
    {
        total += budget2[i] + individuala * budget1[i] / suma;
        newBudget[i] = budget2[i] + individuala * budget1[i] / suma;
    }
    else{
        total += budget1[i] + individualb * budget2[i] / sumb;
        newBudget[i] = budget1[i] + individualb * budget2[i] / sumb;
    }
}
uint z = 0;
while(total < 100)
{
    newBudget[z] += 1;
    total +=1;
    if(z == budget1.length - 1)
    {
        z = 0;
    }
    else
    {
        z ++;
    }
}

return newBudget;
}

```

Again I tried to put as many variables I was using from storage multiple times on the stack and memory variables to save gas cost.

One problem I faced was that there is no built in decimal types in solidity. This

is why my calculate utility has been scaled up by 100 times, it is now between 0 and 10,000. The larger the scale the more accurately I will be able to calculate utility. It also allows more granularity in coalition size factor and coalition size factor increase. Another problem caused by this is that when negotiating a budget between representative and coalition, the total budget would not sum up to 100. To solve this at the end of negotiating it would give each sink one more unit in the budget until it summed up to 100. This causes the problem that earlier sinks will benefit from this much more than later. However smart contracts are deterministic and there is no way to randomise the orders.

Another implantation decision was to decide the order in which representatives take turns. I decided to leave the order of highest weight to lowest as it was. This was for two reason. First it would cost to reorder it, and because letting the biggest representative form coalitions first should end the negotiation earlier and save more gas. I also created another function `calculate100utility`. This returns the utility if the preferred budget and budget are the same. This saved gas as I need this often, and finding out how much the preferred budget overlaps with the budget is expensive.

The negotiation function returned the utility for each agent. After finding the negotiation which returned the best utility for the representative, I then had to create a function `negoToBudget` which took the utility and the representative and coalition budgets and returned the new coalition budget. To do this first, I took each utility and increased it to what it would be if the coalition size was the quota. I then took each of them away from how much the budgets overlap. This gave me three uints, `shardedab` which is how much of the budgets overlap, `individuala` which is how much of the negotiated budget is only in the representative preferred budget, and `individualb` which is how much of the negotiated budget is only the coalitions current budget.

After working out these 3 variables, the function then starting the creating the new budget. First, all the overlapping budget is wanted by both agents so the new budget has them as well. Then I had to decide how the individual a and b where spread. Due to the way I created my utility function, if two sinks are 20 units below the budget, it generated the same utility as one sink 40 units below the budget. I decided to spread the individual utility proportionally by how far away the other agents budget was from the agents.

5.6 Club Contract Website

Chapter 6

Testing

Chapter 7

Conclusions

The System produced solves the current problem with clubs, where all the money is collected and spent by one or a few people. This problem is made worse at university clubs, where it is likely that a first year student has never seen the club before and therefore has no trust in the club or its leadership.

My solution is a smart contract which is run on the Ethereum network. Users can register, vote and decide the budget by sending transactions to the contract. Users do not have to trust the integrity of the smart contract due to the Ethereum miners who validate every transaction.

One problem added was that due to the complete transparency, malicious users could use other users transaction to manipulate the result of an election or final budget. I had to choose between simple algorithms that could be easily manipulated and cheaply run or algorithms that were hard to manipulate but cost more to run. I choose the second option, but tried to reduce the gas price as much as possible.

My system can scale well for users registering and voting. However number of representatives and candidates must be kept low (around 15) or the gas price starts to become too high. However for the last couple of months the gas to use value has been trending down and so in the future may allow for more representatives. I have also written a html and javascript interface for my smart contract. This allows users to interact with the smart contract without a command line.

Chapter 8

Future Work

One problem with my current system, is the html interface, although functional and more accessible than the command line, is still very basic and is not easy to use.

Another part to add to the system is the ability to raise and distribute money during the year with a subset of users e.g some players in the team are going to a tournament. Now all money is raised at the start of the year by everyone and then distributed to managers.

Another addition would be a way to monitor the managers of the sinks. Instead of the all the money being sent to the managers, managers would have to start a vote (maybe only among representatives), and ask for a certain amount of money for some task. At the moment managers could get the money at the start of the year and just leave.

Another problem is that users are all cleared at the end of the term, and then they can register again for the next year. This is a problem as gas is wasted deleting then storing their address and name again. This could be solved with a reregister function.

Also if the second order Copeland algorithm or my budgeting algorithm can be verified in less time than it takes to solve, then the system can scale much better and use a lot less gas. Then users read the input, solve it on their computers, and upload the correct result to the smart contract. The smart contract can then verify the result and if it passes, set the result. However I do not know if this is possible for either of the algorithms

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. [Ethereum alarm clock](#), 2017.

Truffle. [Pet shop tutorial](#), 2017.

Michael Wooldridge. An introduction to multiagent systems, 2009.