DECENTRALIZED AUTONOMOUS ORGANIZATION TO MANAGE A TRUST

FINAL DRAFT - UNDER REVIEW

CHRISTOPH JENTZSCH
FOUNDER & CTO, SLOCK.IT
CHRISTOPH.JENTZSCH@SLOCK.IT

Abstract. This document describes the first implementation of a Decentralized Autonomous Organization (DAO) decentralizing the management of a trust. This concept, as well as the majority robbing the minority attack vector are discussed, and a solution is proposed. Furthermore, a practical implementation of this type of DAO using smart contracts written in Solidity on the Ethereum blockchain is presented and described in detail.

1. Introduction

Historically, the management of trusts has always been centralized. A group of individuals (trustees) would write a contract and combine their resources into one trust, which was then managed by a single person or company, with the trustees losing the power to directly control their money. The rise of crowdfunding (Massolution [2015]), has lowered the barrier of entry to fund projects and has allowed for small contributors to participate in large projects. The money raised still carries the risk of being mismanaged by the company receiving the funding which all too often fails to follow through on its promises or outright disappears without a trace (Knibbs [2015], Biggs [2015]). Thanks to the development of Ethereum (Buterin [2013], Wood [2014]), a blockchain technology which integrates a Turing complete programming language and the ability to process smart contracts, it is now possible to create an organization allowing the trustees to maintain direct control of their funds while still bundling many smaller contributions together to achieve a larger goal. We have written smart contracts (Szabo [1997], Miller [1997]) that form a Decentralized Autonomous Organization (DAO) on the Ethereum blockchain. In this paper we detail the concept of a DAO managing a trust using a concrete example.

After explaining the concept of the DAO, we will cover the "Majority Robbing the Minority Attack" and propose a solution, the "DAO split". Finally, we will explore the smart contracts and conclude with a more detailed specification of the "DAO split".

The code for the smart contracts is located at: `https://github.com/slockit/DAO/`

2. Concept

This section will cover the underlying concept of the DAO and what it is able to do.

After the DAO is deployed on the Ethereum blockchain, anyone is able to fund it by sending ether (the digital fuel powering the Ethereum network) to the address of the DAOs smart contract during a period of time set aside for the initial funding. In exchange, tokens are created which represent membership as well as ownership of a portion of the DAO; these tokens are assigned to the account that sent the ether. The amount of tokens created is proportional to the amount of ether transferred. The price per token varies over time (see section 5). The ownership of these tokens can be transferred to other accounts using transactions on the Ethereum blockchain, as soon as the funding period is over. When deploying the contract, a minimum DAO funding goal and the period of time allowed for the initial funding are chosen. In the event the minimum funding amount is not reached at the end of the funding period, every contributor is refunded their ether. After the funding period has ended, we denote the total ether raised by Ξ_{raised} and the total amount of tokens created by T_{total}. This DAO purely manages funds. In itself, it does not have the capabilities to build a product, write code or develop hardware. It requires a service provider to accomplish these and other goals, which it hires by signing off on a proposal. Every member of the DAO is allowed to submit proposals to spend a portion of the total ether raised, denoted by Ξ_{transfer}, on funding projects. If the proposal is approved, the ether is sent to another smart contract representing the proposed project. Such smart contracts can be parameterized and enable the DAO to interact with and influence the project being funded. An example of such an agreement between the DAO and a project to be funded can be found in the appendix (A.4).

Members of the DAO cast votes weighted by the amount of tokens they control. Tokens are divisible, indistinguishable, fungible and can easily be transferred between accounts. Within the contracts, the individual actions of members, cannot be directly determined. There is a set timeframe t_p to debate and vote on any given proposal. In our example, this timeframe is set by the creator of the proposal, and is required to be at least two weeks for a regular proposal.

After t_p has passed, any token holder can call a function in the DAO contract that will verify that the majority voted in favor of the proposal and that quorum was reached; the function will execute the proposal if this is the case. If this is not the case, the proposal will be closed. The minimum quorum represents the minimum number of tokens required for a vote to be valid, is denoted by q_{min}, and calculated as follows:

$$q_{min} = \frac{T_{\text{total}}}{d} + \frac{\Xi_{\text{transfer}} \cdot T_{\text{total}}}{3 \cdot (\Xi_{DAO} + R_{\text{total}})} \tag{1}$$

Where d is the `minQuorumDivisor`. This parameters default value is 5, but it will double in the case the quorum has not been met for over a year. Ξ_{DAO} is the

amount of ether owned by the DAO and R_{total} is the total amount of reward token as explained in section 7 (also see `totalRewardToken` in A.3). The sum $\Xi_{DAO} + R_{\text{total}}$ is equal to the amount of ether raised plus the rewards received.

This means, initially a quorum of 20% of all tokens is required for any proposal to pass. In the event Ξ_{transfer} equals the amount of ether raised during the funding period plus the rewards received, then a quorum of 53.33% is required.

In order to prevent proposal spam, a minimal deposit is required to be paid when creating a proposal, which gets refunded if quorum is achieved, if quorum is not achieved, the proposal deposit stays with the DAO. The value of the proposal deposit can be changed from the default value by the DAO through another proposal.

3. Notation

Throughout this paper, Ξ always represents an amount of ether in its base unit wei. Which is defined as 1 Wei = 10^{-18} Ether (Wood [2014]). Similarly, DAO tokens are denoted with T and always represent the amount of DAO tokens in its base unit. Which we define as 10^{-16} DAO token.

4. Majority robs minority attack

A problem every DAO has to mitigate is the ability for the majority to rob the minority. An attacker with 51% of the tokens, acquired either during the funding period or purchased afterwards, could make a proposal to send all the funds to themselves. Since they would hold the majority of the tokens, they would always be able to pass their proposals.

To prevent this, the minority always needs the option to retrieve their portion of the funds. This is done by splitting the DAO into two. In the case an individual, or a group of token holders, strongly disagree with a proposal and want to retrieve their portion of the funds before the proposal gets executed, they can submit a special type of proposal to form a new DAO. The minority can then vote to move their funds to this new DAO, leaving the majority alone only able to spend their own money. This idea originates from a blog post by Vitalik Buterin (Buterin [2015]).

A problem this simple fix doesnt address is voter apathy: some token holders might not be actively involved in the DAO and might not follow proposals closely. An attacker could use this to their advantage. Even though the minority has the chance to retrieve their funds and split the DAO, some of them could be unaware of the situation and won't do this. For the DAO to be considered safe, it is required that inactive token holders will not lose their funds. Our proposed solution is implemented by limiting each individual DAO to a single service provider. This service provider controls the only account that can receive ether from the DAO, across all proposals. In addition, the service provider can build a whitelist of addresses that the DAO is allowed to send funds to. This gives the service provider of the DAO considerable power. To prevent the abuse of this power, it is possible for the DAO to vote for a new service provider, which may result in a split of the DAO as described above.

Any token holder can make a proposal to vote for a new service provider. In effect, even a single token holder is able to both retrieve their remaining portion of ether and maintain their right to any future rewards associated to their previous contribution, as these will be sent to the new DAO automatically. Rewards are defined as any ether received by the DAO generated from products the DAO funded so far and are explained in further detail in section 7.

The process of choosing a new service provider is as follows: Any token holder can start a proposal to vote for a new service provider. For this proposal there is no proposal deposit to be paid, otherwise the attacker could vote for an extremely high deposit, preventing any splits. The debating period for this proposal is 10 days. This is 4 days less than the minimum required for regular proposals, allowing anyone to retrieve their funds before a potentially malicious proposal goes through. There is no quorum requirement, so that every token holder has the ability to split into their own DAO. The debating period is used to discuss the new service provider and conduct an informal vote. The result of the vote has no actual consequence, it is purely instructional. After this first voting round, token holders can cast a second vote to confirm the result. In the case a majority opts to keep the original service provider, the minority can either change their vote to keep the original service provider in order to avoid a split, or inversely they can confirm their vote for a new service provider and move their portion of the funds to a new DAO.

5. Token Price

In order to reward contributors who purchase tokens early during the funding period, since they had less information, and hence a greater risk, than contributors who joined later, they will pay less than contributors who joined later. For the DAO described here, we have chosen the following price schedule:

(2)
$$P(t) = \begin{cases} 0.01 & \text{if} \quad t < t_c - 2 \cdot \mathrm{w} \\ 0.01 + 0.0005 \cdot m(t) & \text{if} \quad t_c - 2 \cdot \mathrm{w} \leqslant t < t_c - 4 \cdot \mathrm{d} \\ 0.015 & \text{otherwise} \end{cases}$$

with the multiplier m defined as:

(3)
$$m(t) = (t - (t_c - 2 \cdot \mathrm{w}))/\mathrm{d}$$

Here t is the unix time in seconds, t_c is the closing time of the funding period (see A.2 `closingTime`), w is a week in seconds and d a day in seconds. Hence the number of tokens (in its base unit) each buyer receives is calculated as: $P(t) \cdot \Xi_c$. Here Ξ_c stands for the amount of ether payed, denoted in wei. This results in a constant price in the beginning, until 2 weeks before the end of the token sale. At this time the price increased daily by $0.005\,\Xi_c$ per base unit of DAO token. Until 4 days before the closing time when there will be a constant price of $0.015\ \Xi_c$ per base unit of DAO token.

This price increase leads to a situation where a single contributor, who has bought tokens during the initial price, can split the DAO right after the end of the sale and would get more ether out than they put in due to other contributors paying a higher price (Green [2016]). In order to avoid that possibility, all ether that is payed above the

initial price of the token, will be sent to an extra account. Denoted as `extraBalance` in A.2. This money can be sent back to the DAO through a proposal after the DAO has spent at least this amount of ether. This rule is implemented in the internal function `isRecipientAllowed` in section 6.3.

6. Contracts

This section will detail the smart contracts implementing the aforementioned concept. The contracts are written in the programming language Solidity (Reitwiessner and Wood [2015]). Each contract has member variables and functions which can be externally called by sending a transaction to the Ethereum network with the DAO contract address as the recipient and the method ID (optional with parameters) as data. In this section we will discuss the meaning of the variables and the functions in detail.

The main contract is called 'DAO'. It defines the inner workings of the DAO and it derives the member variables and functions from 'Token' and 'TokenSale'. Token defines the inner workings of the DAO Token and TokenSale defines how the DAO token is purchased for ether. In addition to these three contracts, there is the 'ManagedAccount' contract, which acts as a helper contract to store the rewards which are to be distributed to the token holders and the `extraBalance` (see section 5). The contract 'SampleOffer' (A.4) is an example of what a proposal from the service provider to the DAO could look like.

6.1. Token.

```
contract TokenInterface {
    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;
    uint256 public totalSupply;
    function balanceOf(address _owner) constant returns (uint256 balance);
    function transfer(address _to, uint256 _amount) returns (bool success);
    function transferFrom(address _from, address _to, uint256 _amount) returns (bool success);
    function approve(address _spender, uint256 _amount) returns (bool success);
    function allowance(address _owner, address _spender) constant returns (uint256 remaining);

    event Transfer(address indexed _from, address indexed _to, uint256 _amount);
    event Approval(address indexed _owner, address indexed _spender, uint256 _amount);
}
```

Above is the interface of the Token contract. The interfaces of these contracts are used in the text of this document to give a simple overview of the functions and variables used in the contract, the full implementation can be found in the appendix (A.1). This contract represents the standard token as described here: `https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs`, and the contract `https://github.com/ConsenSys/Tokens/blob/master/Token_Contracts/contracts/Standard_Token.sol` was used as a starting point for the contracts creation.

The map `balances` stores the number of DAO tokens which are controlled by an address. All contracts which derive from `TokenInterface` can directly modify this map, but only 4 functions do so: `buyTokenProxy`, `transfer` , `transferFrom` and `splitDAO`.

The map `allowed` is used to track the previously specified addresses that are allowed to send tokens on someone else's behalf.

The integer `totalSupply` is the total number of DAO tokens in existence. The `public` keyword creates a function with the same name as the variable which returns its value so that it is publically available.

The function `balanceOf` returns the balance of the specified address.

The function `transfer` is used to send token from the sender of the message to another address.

The function `transferFrom` is used to transfer tokens on behalf of someone else who has approved the transfer in advance using the `approve` function.

The function `approve` can be used by the DAO token owner to specify a certain `spender` to transfer a specified `value` from their account using the `transferFrom` function. To check whether a certain address is allowed to spend DAO tokens on behalf of someone else, the `allowance` function can be used, which returns the number of tokens which can be spent by the `spender`. This is similar to writing a check.

The event `Transfer` is used to inform lightweight clients about changes in `balances`.

The event `Approval` is used to inform lightweight clients about changes in `allowed`.

6.2. TokenSale.

```
contract TokenSaleInterface {
    uint public closingTime;
    uint public minValue;
    bool public isFunded;
    address public privateSale;
    ManagedAccount extraBalance;
    mapping (address => uint256) weiGiven;
```

```
    function TokenSale(uint _minValue, uint _closingTime);
    function buyTokenProxy(address _tokenHolder) returns (bool success);
    function refund();
    function divisor() returns (uint divisor);

    event FundingToDate(uint value);
    event SoldToken(address indexed to, uint amount);
    event Refund(address indexed to, uint value);
}
```

Above is the interface of the `TokenSale` contract (A.2).

The integer `closingTime` is the (unix) time at which the token sale ends.

The integer `minValue` is the number of wei which are needed to be received by the DAO in order to get funded.

The boolean `isFunded` is true if DAO has reached its minimum funding goal, false otherwise.

The address `privateSale` is used for DAO splits - if it is set to 0, then it is a public sale, otherwise, only the address stored in `privateSale` is allowed to purchase tokens.

The managed account (A.5) `extraBalance` is used to hold the excess ether which is received after the token price is increased during the sale. Anything that has been paid above the initial price goes to this account.

The map `weiGiven` stores the amount of wei given by each contributor during the sale and is only used to refund the contributors if the Token sale does not reach its funding goal.

The constructor `TokenSale` initiates the token sale with the arguments `minValue`, `closingtime` and `privateSale`, which will be set in the constructor of the DAO contract (A.3) which is only executed once, when the DAO is deployed.

The function `buyTokenProxy` creates one unit of the DAO tokens minimum denomination for every wei sent. The price is calculated as

$$\Xi_c \cdot 20/\text{divisor}$$

Here Ξ_c is the amount of wei given in order to purchase tokens, and `divisor` is calculated depending on the time as described in section 5.

The parameter `tokenHolder` defines the receiver of the newly minted tokens.

The function `refund` can be called by any contributor to recieve their wei back if the token sale failed to meet its funding goal.

The function `divisor` is used to calculate the price of the token during the sale in the function `buyTokenProxy`.

The events `FundingToDate`, `SoldToken` and `Refund` are used to inform lightweight clients of the status of the token sale.

6.3. **DAO.**

```
contract DAOInterface {
    Proposal[] public proposals;
    uint minQuorumDivisor;
    uint lastTimeMinQuorumMet;
    uint public rewards;
    address[] public allowedRecipients;
    mapping (address => uint) public rewardToken;
    uint public totalRewardToken;
    ManagedAccount public rewardAccount;
    mapping (address => uint) public paidOut;
    mapping (address => uint) public blocked;
    uint public proposalDeposit;
    DAO_Creator public daoCreator;

    struct Proposal {
        address recipient;
        uint amount;
        string description;
        uint votingDeadline;
        bool open;
        bool proposalPassed;
        bytes32 proposalHash;
        uint proposalDeposit;
        bool newServiceProvider;
        SplitData[] splitData;
        uint yea;
        uint nay;
        mapping (address => bool) votedYes;
        mapping (address => bool) votedNo;
```

```
    address creator;
}

struct SplitData {
    uint splitBalance;
    uint totalSupply;
    uint rewardToken;
    DAO newDAO;
}

modifier onlyTokenholders {}

function DAO(
    address _defaultServiceProvider,
    DAO_Creator _daoCreator,
    uint _minValue,
    uint _closingTime,
    address _privateSale
)
function () returns (bool success);
function payDAO() returns(bool);
function receiveEther() returns(bool);
function newProposal(
    address _recipient,
    uint _amount,
    string _description,
    bytes _transactionData,
    uint _debatingPeriod,
    bool _newServiceProvider
) onlyTokenholders returns (uint _proposalID);
function checkProposalCode(
    uint _proposalID,
    address _recipient,
    uint _amount,
    bytes _transactionData
) constant returns (bool _codeChecksOut);
function vote(
    uint _proposalID,
    bool _supportsProposal
) onlyTokenholders returns (uint _voteID);
function executeProposal(
    uint _proposalID,
    bytes _transactionData
) returns (bool _success);
function splitDAO(
    uint _proposalID,
    address _newServiceProvider
) returns (bool _success);
function addAllowedAddress(address _recipient) external returns (bool _success);
function changeProposalDeposit(uint _proposalDeposit) external;
function getMyReward() returns(bool _success);
function withdrawRewardFor(address _account) returns(bool _success);
function transferWithoutReward(address _to, uint256 _amount) returns (bool success);
function transferFromWithoutReward(
    address _from,
    address _to,
    uint256 _amount
) returns (bool success);
function halveMinQuorum() returns (bool _success);
function numberOfProposals() constant returns (uint _numberOfProposals);
function isBlocked(address _account) returns (bool);

event ProposalAdded(
```

```
        uint indexed proposalID,
        address recipient,
        uint amount,
        bool newServiceProvider,
        string description
    );
    event Voted(uint indexed proposalID, bool position, address indexed voter);
    event ProposalTallied(uint indexed proposalID, bool result, uint quorum);
    event NewServiceProvider(address indexed _newServiceProvider);
    event AllowedRecipientAdded(address indexed _recipient);
}
```

The original contract used as a starting point for DAO was: `http://chriseth.github.io/browser-solidity/?gist=192371538cf5e43e6dab` as described in `https://blog.ethereum.org/2015/12/04`. The main feature added is the splitting mechanism and all that comes with it. Now we will define the member variables and functions one at a time.

The array `proposals` holds all the proposals ever made.

The integer `minQuorumDivisor` is used to calculate the quorum needed for a proposal to pass. It is set to 5, but will double in the case a quorum has not been reached for over a year.

The integer `lastTimeMinQuorumMet` keeps track of the last time the quorum was reached.

The integer `rewards` counts all the rewards which have been sent to the DAO. It gets set to zero when paid out to the `rewardAccount`.

The address `serviceProvider` is set at the creation of the DAO and defines the service provider.

The list `allowedRecipients` is commonly referred to as the whitelist. The DAO can only send transactions to itself, the `serviceProvider`, `rewardAccount`, `extraBalance` and addresses in the whitelist. Only the `serviceProvider` can add addresses to the whitelist.

The map `rewardToken` tracks the addresses that are owed rewards generated by the products of the DAO. Those addresses can only be DAOs.

The integer `totalRewardToken` tracks the amount of reward tokens in existence.

The variable `rewardAccount` is of the type `ManagedAccount` , which will be discussed in A.5. It is used to manage the rewards which are to be distributed to the DAO Token Holders and reward token holders.

The map `paidOut` is used to keep track how much wei a single token holder has already retrieved from `rewardAccount`.

The map `blocked` stores the addresses of the DAO Tokens that have voted and therefore are not allowed to be transferred until the vote has concluded.. The address points to the proposal ID.

The integer `proposalDeposit` specifies the minimum deposit to be paid in wei for any proposal that does not include a change in service providers.

The contract `daoCreator` is used to create a new DAO with the same code as this DAO, used in the case of a split.

A single proposal has the parameters:

recipient: The address where the `amount` of wei will go to if the proposal is accepted.

amount: The amount of wei to transfer to `recipient` if the proposal is accepted.

description: A plain text description of the proposal.

votingDeadline: A unix timestamp, denoting the end of the voting period.

open: A boolean which is false if the votes have already been counted, true otherwise.

proposalPassed: A boolean which is true if a quorum has been achieved with the majority approving the proposal.

proposalHash: A hash to check validity of a proposal. Equal to `sha3(_recipient, _amount, _transactionData)`.

proposalDeposit: The minimum deposit (in wei) the creator of a proposal is required to send to submit a proposal. It is taken from the `msg.value` of a `newProposal` call; its purpose is to prevent spam. It is set to 20 ether by default but the creator of the proposal can send any amount above this for the deposit.For Slock.its purposes, the proposals will be sorted by the proposal deposit in the GUI, so in the case that a proposal is considered important, the creator of the proposal can deposit extra ether to advertise their proposal. The creator of the proposal will be refunded the entire deposit if quorum is reached, if it is not reached the deposit remains with the DAO.

newServiceProvider: A boolean which is true if this proposal assigns a new service provider.

splitData: The data used to split the DAO. This data is gathered from proposals when they require a new service provider.

yea: Number of tokens in favor of the proposal.

nay: Number of tokens in opposed to the proposal.

votedYes: Simple mapping to check if a token holder has voted for it.

votedNo: Simple mapping to check if a token holder has voted against it.

creator: The address of the token holder that created the proposal.

The split data structure is used to split the DAO. It contains:

splitBalance: The balance of the current DAO minus the proposal deposit at the time of split.

totalSupply: The total amount of DAO tokens in existence at the time of the split.

rewardToken: The amount of reward tokens owned by the original DAO at the time of split.

newDAO: The address of the new DAO contract (0 if not created yet).

Those are all the member variables which are stored in this smart contract on the blockchain. This information can at any time be read from the blockchain using an Ethereum client.

We will now discuss the functions of the DAO contract in detail. Many of the member variables that are used in this contract are defined in one of the other three contracts.

There is a special function which is called the constructor. It has the same name as the contract "DAO". This function is only executed once, when the DAO is created. In the DAO constructor, the following variables are set:

- `serviceProvider`
- `daoCreator`
- `proposalDeposit`
- `rewardAccount`
- `minValue`
- `closingTime`
- `privateSale`

In order to interact with the smart contract the following functions are used:

fallback function. The fallback function is a function without a specific name. It is called when the contract receives a transaction without data (a pure value transfer). There are no direct arguments for this function. The fallback function will call `buyTokenProxy` passing the address of the sender as an argument during the token sale. This will trigger the immediate purchase of tokens. In order to protect users, this function will send the ether received after the end of the sale back to the sender for a time period of 40 days. After which this function is repurposed to receive ether as simple deposit to the DAO using the function `receiveEther`.

payDAO. This function is used to receive and track the rewards generated by the products funded by the DAO and returns true when ether has been added to `rewards`. In the context of Slock.it, these will be the rewards generated when Slocks are deployed or used.

receiveEther. A simple function used to receive ether. It does nothing but return true when the DAO receives ether.

newProposal. This function is used to create a new proposal. The arguments of the function are:

recipient: The address of the recipient of the ether in the proposal (has to be the DAO address itself, the current service provider or an address on the whitelist `allowedRecipients`).
amount: The amount of wei to be sent in the proposed transaction.
description: A string describing the proposal.
transactionData: The data of the proposed transaction.
debatingPeriod: The amount of time to debate the proposal, at least 2 weeks for a normal proposal and at least 10 days for a new service provider proposal.
newServiceProvider: A boolean defining whether this proposal is for a new service provider or not.

After checking the sanity of the proposal (see code), this function creates a proposal which is open for voting for a certain amount of time. The function will return a proposal ID which is used to vote.

checkProposalCode. This function is used to check that a certain proposal ID matches a certain transaction. The arguments of the function are:

proposalID: The proposal ID.
recipient: The address of the recipient of the proposed transaction.
amount: The amount of wei to be sent with the proposed transaction.
transactionData: The data of the proposed transaction.

If the `recipient`, the `amount` and the `transactionData` match the proposal ID, the function will return `true` , otherwise it will return `false`. This will be used to verify that the proposal ID matches what the DAO token holder thinks they are voting on.

vote. This function is used to vote on a proposal. The arguments of the function are:

proposalID: The proposal ID.
supportsProposal: A boolean Yes/No does the DAO token holder support the proposal

The function simply checks whether the sender has yet to vote and whether the proposal is still open for voting. If both requirements are met, it records the vote in the storage of the contract. The tokens used to vote will be blocked, meaning they can not be transferred until the proposal is closed.This is to avoid voting several times with different sender addresses.

executeProposal. This function can be called by anyone. It counts the votes, in order to check whether the quorum is met, and executes the proposal if it passed, unless it is a proposal for a new service provider, than it does nothing. The arguments of the function are:

proposalID: The proposal ID.
transactionData: The data of the proposed transaction

The function checks whether the voting deadline has passed and that the `transactionData` matches the proposal ID. Then it checks whether the quorum has been met (see Eq. 1) and if the proposal had a majority of support. If this is the case, it executes the proposal and refunds the proposal deposit. If the quorum has been achieved, but the proposal was declined by the majority of the voters, the proposal deposit is refunded and the proposal closes.

splitDAO. After a new service provider has been proposed, and the debating period in which the token holders could vote for or against the proposal has passed, this function is called by each of the DAO token holders that want to leave the current DAO and move to a new DAO with the proposed new service provider. This function creates a new DAO and moves a portion of the ether, as well as a portion of the reward tokens to the new DAO. The arguments are:

proposalID: The proposal ID.
newServiceProvider: The address of the new service provider of the new DAO.

After a sanity check (see code), this function will create the new DAO if it hasnt already been created using the contract `daoCreator`, updates the split data stored in the proposal and stores the address of the new DAO in the split data. This function moves the portion of ether that belongs to the caller of this function in the original DAO to the new DAO. This ether amount is denoted by Ξ_{sender}, stated in wei and is calculated as follows:

$$\Xi_{\text{sender}} = \Xi_{\text{DAO}} \cdot T_{\text{sender}}/T_{\text{total}} \tag{4}$$

Here T_{sender} is the amount of tokens of the caller of the function and Ξ_{DAO} is the balance of the DAO at the time of the split. This will be used to effectively buy tokens of the newly created DAO and fund the new DAO just as the original DAO was funded. In addition to the ether which is moved to the new DAO, the reward tokens R_{sender} are also transferred. They are calculated as follows:

$$R_{\text{sender}} = R_{\text{DAO}} \cdot T_{\text{sender}}/T_{\text{total}} \tag{5}$$

Where R_{DAO} is the amount of reward tokens owned by the original DAO at the time of the split. These tokens allow the new DAO to retrieve their portion of the reward using the `getMyReward` function of the original DAO. At the end of this process all original DAO tokens of the sender account are destroyed.

transfer and transferFrom. These functions overload the functions defined in the `Token` contract. They do call `transfer / transferFrom` function in the `Token` contract, but they additionally transfer information about the already paid out rewards attached to the tokens being transferred using the `transferPaidOut` function.

transferPaidOut. This function is called to when making any transfer of DAO tokens using `transfer` or `transferFrom` and it updates the array `paidOut` to track the amount of rewards which has been paid out already, P, and is calculated as follows:

$$P = P_{\text{from}} \cdot T_{\text{amount}}/T_{\text{from}} \tag{6}$$

Here P_{from} is the total amount of ether which has been paid out to the `from` address (the sender), T_{amount} is the amount of tokens to be transferred and T_{from} the amount of tokens owned by the `from` address.

transferWithoutReward and transferFromWithoutReward. The same as `transfer` and `transferFrom`, but it calls `getMyReward` prior to that.

getMyReward. Calls `withdrawRewardFor` with the sender as the parameter. This is used to withdraw the portion of the rewards which belong to the sender from the `rewardAccount`.

withdrawRewardFor. This function is used to retrieve the portion of the rewards in the `rewardAccount` which belong to address given in the parameter. Their portion of the reward tokens S is calculated as follows:

$$S = R_{\text{DAO}} \cdot T_{\text{sender}}/T_{\text{total}} + R_{\text{sender}} \tag{7}$$

Here R_{DAO} is the amount of reward tokens owned by the DAO, and T_{sender} the amount of DAO tokens which are directly owned by the sender. R_{sender} will always be zero except in the case that the sender is a DAO that has split from the original DAO and owns their own reward tokens from the split. The amount of ether Ξ_{reward} which is then sent to the DAO token holder that calls this function is:

$$\Xi_{\text{reward}} = \Xi_{\text{rewardAccount}} \cdot S/R_{\text{total}} - \Xi_{\text{paidOut[sender]}} \tag{8}$$

Here $\Xi_{\text{rewardAccount}}$ is the total rewards ever received by the `rewardAccount`, R_{total} is the total amount of reward tokens that have ever been created, regardless of splits (`totalRewardToken`) and $\Xi_{\text{paidOut[sender]}}$ is the total amount of wei which has already been paid out to the DAO token holder calling the function. The reward tokens are further elaborated in section 8.

addAllowedAddress. This function adds an address to the whitelist, `allowedRecipients`. It can only be executed by the service provider.

halveMinQuorum. When called, halves the minimum quorum in the case it has not been met for over 52 weeks, by doubling `minQuorumDivisor`.

numberOfProposals. Returns the total number of proposals ever created.

isBlocked. This function returns true when the address given as parameter is currently blocked to transfer tokens due to an ongoing vote it has participated in, otherwise it returns false.

changeProposalDeposit. This function changes the parameter `proposalDeposit`. It can only be called by the DAO through a transaction which was proposed and voted for by a majority of the token holders.

6.4. Managed Account.

```
contract ManagedAccountInterface {
    address public owner;
    uint public accumulatedInput;

    function payOut(address _recipient, uint _amount) returns (bool);

    event PayOut(address _recipient, uint _amount);
}
```

This contract is used to manage the rewards and the `extraBalance` (as explained in section 5). It has two member variables:

The address `owner`, is the only address with permission to withdraw from that account (in our case the DAO) and send ether to another address using the `payOut` function.

The integer, `accumulatedInput`, represents the total sum of ether (in wei) which has been sent to this contract so far.

The fallback function is called when the contract receives a transaction without data (a pure value transfer). There are no direct arguments for this function. When

it is called it counts the amount of ether it receives and stores it in `accumulatedInput`.

The function `payOut` can only be executed by the `owner` (in our case the DAO). It has two arguments: `recipient` and `amount`. It is used to send `amount` wei to a `recipient` and is called by `getMyReward` in the DAO contract.

7. Reward Tokens

In this section we give a description of how reward tokens are implemented in this contract. Much of the information has already been explained, but it is restated here for clarity.

Reward tokens are used to divide the ether sent to `rewardAccount` amongst the various DAOs that own reward tokens. Reward tokens are only transferred in the event of a DAO split, they can never be owned by anything other than the original DAO or a fork of the original DAO that generated the reward tokens.

Reward tokens are generated when the DAO makes any transaction spending ether (with the exception of sending ether to the `rewardAccount`). When the DAOs products send ether back to the DAO (for example when a Slock sends 1% of a transaction to the DAO), the ether is held with the rest of the ether that the DAO has, but `rewards` counts all the ether which has been received as rewards. The DAO can use these rewards to fund new proposals or to equally distribute the rewards to the reward token holders (using a proposal which gets voted on by the DAO token holders). Then the token holders of the DAOs that hold reward tokens will be able to claim the ether they have earned for their contribution to the original DAO that issued the reward token. To do this the DAO will send the accumulated rewards to the `rewardAccount` which is held in the `ManagedAccount` contract. Then and only then will the token holders of the DAOs that hold reward tokens be able to call the `getMyReward` function and receive their ether. These payouts are tracked by the map `paidOut` which keeps track of which DAOs hold reward tokens and whether they have claimed their portion of the rewards, and which of the original DAO token holders have claimed their fair portion of the rewards.This process guarantees that any DAO token holder whose ether was spent building a product will receive the rewards promised to them from that product even if they decide to split from the DAO.

8. Split

In this section we want to formally describe some parameters and their behavior during a split.

The total amount of DAO tokens `totalSupply` is defined as follows:

$$T_{\text{total}} = \sum_{i=0}^{2^{256}-1} T_i \tag{9}$$

Where T_i is the amount of DAO tokens owned by an address i (`balances[i]`). Note that 2^{256} is the total number of possible addresses in Ethereum. Similarly, the amount of reward tokens R_{total} is defined as follows:

$$R_{\text{total}} = \sum_{i=0}^{2^{256}-1} R_i = \sum_{p=0;p.proposalPassed=true}^{numProposals} p.amount \tag{10}$$

For every passed proposal that sends ether out of the DAO, an amount of reward tokens equal to the amount being spent (in wei) is created.

Lets assume that during the split, a fraction of DAO tokens, X, changes service providers and leaves the DAO. The new DAO created receives $X \cdot \Xi_{\text{DAO pre}}$, a portion of the remaining ether from the original DAO.

$$\Xi_{\text{DAO post}} = (1 - X) \cdot \Xi_{\text{DAO pre}} \tag{11}$$

Here $\Xi_{\text{DAO pre}}$ is the ether balance of the original DAO before the split and $\Xi_{\text{DAO post}}$ is the ether balance of the original DAO after the split.

A portion of the reward tokens is transferred to the new DAO in a very similar manner:

$$R_{\text{DAO post}} = (1 - X) \cdot R_{\text{DAO pre}} \tag{12}$$

Here R_{DAO} is the amount of reward tokens owned by the DAO (prior to the first split 100% of all rewards tokens ever created are owned by the DAO).

$$R_{\text{newDAO}} = (X) \cdot R_{\text{DAO pre}} \tag{13}$$

The number of reward tokens owned by the new DAO are denoted by R_{newDAO}. The total amount of reward tokens R_{total} stays constant during the split, no reward tokens are ever destroyed.

The original DAO tokens of the accounts that confirmed the new service provider are destroyed. Hence:

$$T_{\text{total post}} = (1 - X) \cdot T_{\text{total pre}} \tag{14}$$

This process allows DAO token holders to retrieve their ether from the DAO at any time without losing out on any of the future rewards they are entitled to receive even if they choose to leave the DAO.

9. Updates

Although the code of the contract specified at a certain address in the Ethereum blockchain can not be changed, there might still be a need for a single member or the DAO as a whole to change the contracts. Every single member can always split the DAO as described above and move their funds to a new DAO. From there they can move their funds to another new DAO with a new smart contract. But in order to use a new code for the complete DAO one can simply create a new DAO contract with all the needed features and deploy it on the blockchain, and make a proposal to send all the ether in the old DAO to this contract. If accepted, the complete DAO moves to the new contract. In order to use the same underlying DAO tokens there, one can use the `approve` function and give the new DAO the right to move the tokens. In the new contract this right should only be usable in restricted functions which are only callable by the owner of the tokens. This process allows for the DAO to maintain static immutable code on the Ethereum blockchain, while still being able to be updated if the need arises.

10. Acknowledgements

I want to thank Stephan Tual and Simon Jentzsch for fruitful discussions and corrections, as well as Gavin Wood and Christian Reitwiessner for a review of the contracts and the development of Solidity, the programing language used to write the contracts.

Special thanks goes to Yoichi Hirai and Lefteris Karapetsas for reviewing the smart contracts and making significant improvements.

I also want to thank Griff Green for reviewing and editing the paper.

Last but not least I want to thank our community which has given feedback, corrections and encouragement.

References

John Biggs. When Crowdfunding Fails The Backers Are Left With No Way Out. 2015. URL http://techcrunch.com/2015/11/19/when-crowdfunding-fails-the-backers-are-left-with-no-way-out/.

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. 2013. URL https://github.com/ethereum/wiki/wiki/White-Paper.

Vitalik Buterin. The Subjectivity / Exploitability Tradeoff. 2015. URL https://blog.ethereum.org/2015/02/14/subjectivity-exploitability-tradeoff/.

Griff Green. private discussion. 2016.

Kate Knibbs. The 9 Most Disgraceful Crowdfunding Failures of 2015. 2015. URL http://gizmodo.com/the-9-most-disgraceful-crowdfunding-failures-of-2015-1747957776.

Massolution. 2015CF - Crowdfunding Industry Report. 2015. URL http://reports.crowdsourcing.org/index.php?route=product/product&path=0_20&product_id=54.

Mark Miller. The Future of Law. In *paper delivered at the Extro 3 Conference (August 9)*, 1997.

Christian Reitwiessner and Gavin Wood. Solidity. 2015. URL http://solidity.readthedocs.org/.

Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. 2014. URL http://gavwood.com/paper.pdf.

Appendix A. Contracts

A.1. **Token.**

```
contract TokenInterface {
    mapping (address => uint256) balances;
    mapping (address => mapping (address => uint256)) allowed;

    /// @return Total amount of tokens
    uint256 public totalSupply;

    /// @param _owner The address from which the balance will be retrieved
    /// @return The balance
    function balanceOf(address _owner) constant returns (uint256 balance);

    /// @notice Send `_amount` tokens to `_to` from `msg.sender`
    /// @param _to The address of the recipient
    /// @param _amount The amount of tokens to be transferred
    /// @return Whether the transfer was successful or not
    function transfer(address _to, uint256 _amount) returns (bool success);

    /// @notice Send `_amount` tokens to `_to` from `_from` on the condition it
    /// is approved by `_from`
    /// @param _from The address of the sender
    /// @param _to The address of the recipient
    /// @param _amount The amount of tokens to be transferred
    /// @return Whether the transfer was successful or not
    function transferFrom(address _from, address _to, uint256 _amount)
        returns (bool success);

    /// @notice `msg.sender` approves `_spender` to spend `_amount` tokens on
    /// its behalf
    /// @param _spender The address of the account able to transfer the tokens
    /// @param _amount The amount of tokens to be approved for transfer
    /// @return Whether the approval was successful or not
    function approve(address _spender, uint256 _amount) returns (bool success);

    /// @param _owner The address of the account owning tokens
    /// @param _spender The address of the account able to transfer the tokens
    /// @return Amount of remaining tokens of _owner that _spender is allowed
    /// to spend
```

```
    function allowance(address _owner, address _spender)
        constant
        returns (uint256 remaining);

    event Transfer(address indexed _from, address indexed _to, uint256 _amount);
    event Approval(
        address indexed _owner,
        address indexed _spender,
        uint256 _amount
    );
}


contract Token is TokenInterface {
    // Protects users by preventing the execution of method calls that
    // inadvertently also transferred ether
    modifier noEther() {if (msg.value > 0) throw; _}

    function balanceOf(address _owner) constant returns (uint256 balance) {
        return balances[_owner];
    }

    function transfer(address _to, uint256 _amount)
        noEther
        returns (bool success)
    {
        if (balances[msg.sender] >= _amount && _amount > 0) {
            balances[msg.sender] -= _amount;
            balances[_to] += _amount;
            Transfer(msg.sender, _to, _amount);
            return true;
        }
        else
           return false;
    }

    function transferFrom(address _from, address _to, uint256 _amount)
        noEther
        returns (bool success)
    {
        if (balances[_from] >= _amount
            && allowed[_from][msg.sender] >= _amount
            && _amount > 0
        ) {
            balances[_to] += _amount;
            balances[_from] -= _amount;
            allowed[_from][msg.sender] -= _amount;
            Transfer(_from, _to, _amount);
            return true;
        }
        else
            return false;
    }

    function approve(address _spender, uint256 _amount) returns (bool success) {
        allowed[msg.sender][_spender] = _amount;
        Approval(msg.sender, _spender, _amount);
        return true;
    }

    function allowance(address _owner, address _spender)
        constant
        returns (uint256 remaining)
```

```
    {
        return allowed[_owner][_spender];
    }
}
```

A.2. **TokenSale.**

```
contract TokenSaleInterface {

    // End of token sale, in Unix time
    uint public closingTime;
    // Minimum funding goal of the token sale, denominated in tokens
    uint public minValue;
    // True if the DAO reached its minimum funding goal, false otherwise
    bool public isFunded;
    // For DAO splits - if privateSale is 0, then it is a public sale, otherwise
    // only the address stored in privateSale is allowed to purchase tokens
    address public privateSale;
    // hold extra ether which has been paid after the DAO token price has increased
    ManagedAccount extraBalance;
    // tracks the amount of wei given from each contributor (used for refund)
    mapping (address => uint256) weiGiven;

    /// @dev Constructor setting the minimum funding goal and the
    /// end of the Token Sale
    /// @param _minValue Token Sale minimum funding goal
    /// @param _closingTime Date (in Unix time) of the end of the Token Sale
    // This is the constructor: it can not be overloaded so it is commented out
    //  function TokenSale(uint _minValue, uint _closingTime);

    /// @notice Buy Token with `_tokenHolder` as the initial owner of the Token
    /// @param _tokenHolder The address of the Tokens's recipient
    function buyTokenProxy(address _tokenHolder) returns (bool success);

    /// @notice Refund `msg.sender` in the case the Token Sale didn't reach its
    /// minimum funding goal
    function refund();

    /// @return the divisor used to calculate the token price during the sale
    function divisor() returns (uint divisor);

    event FundingToDate(uint value);
    event SoldToken(address indexed to, uint amount);
    event Refund(address indexed to, uint value);
}


contract TokenSale is TokenSaleInterface, Token {
    function TokenSale(uint _minValue, uint _closingTime, address _privateSale) {
        closingTime = _closingTime;
        minValue = _minValue;
        privateSale = _privateSale;
        extraBalance = new ManagedAccount(address(this));
    }

    function buyTokenProxy(address _tokenHolder) returns (bool success) {
        if (now < closingTime && msg.value > 0
        && (privateSale == 0 || privateSale == msg.sender)) {
            uint token = (msg.value * 20) / divisor();
            extraBalance.call.value(msg.value - token)();
            balances[_tokenHolder] += token;
            totalSupply += token;
            weiGiven[msg.sender] += msg.value;
            SoldToken(_tokenHolder, token);
```

```
            if (totalSupply >= minValue && !isFunded) {
                isFunded = true;
                FundingToDate(totalSupply);
            }
            return true;
        }
        throw;
    }

    function refund() noEther {
        if (now > closingTime && !isFunded) {
            // get extraBalance - will only succeed when called for the first time
            extraBalance.payOut(address(this), extraBalance.accumulatedInput());

            // execute refund
            if (msg.sender.call.value(weiGiven[msg.sender])()) {
                Refund(msg.sender, weiGiven[msg.sender]);
                totalSupply -= balances[msg.sender];
                balances[msg.sender] = 0;
                weiGiven[msg.sender] = 0;
            }
        }
    }

    function divisor() returns (uint divisor){
        // the number of (base unit) tokens per wei is calculated
        // as `msg.value` * 20 / `divisor`
        // the funding period starts with a 1:1 ratio
        if (closingTime - 2 weeks > now) return 20;
        // followed by 10 days with a daily price increase of 5%
        else if (closingTime - 4 days > now)
            return (20 + (now - (closingTime - 2 weeks)) / (1 days));
        // the last 4 days there is a constant price ratio of 1:1,5
        else return 30;
    }
}
```

A.3. **DAO.**

```
contract DAOInterface {

    // Proposals to spend the DAO's ether or to choose a new service provider
    Proposal[] public proposals;
    // The quorum needed for each proposal is partially calculated by
    // totalSupply / minQuorumDivisor
    uint minQuorumDivisor;
    // The unix time of the last time quorum was reached on a proposal
    uint lastTimeMinQuorumMet;
    // The total amount of wei received as reward that has not been sent to
    // the rewardAccount
    uint public rewards;
    // Address of the service provider
    address public serviceProvider;
    // The whitelist: List of addresses the DAO is allowed to send money to
    address[] public allowedRecipients;

    // Tracks the addresses that own Reward Tokens. Those addresses can only be
    // DAOs that have split from the original DAO. Conceptually, Reward Tokens
    // represent the proportion of the rewards that the DAO has the right to
    // receive. These Reward Tokens are generated when the DAO spends ether.
    mapping (address => uint) public rewardToken;
    // Total supply of rewardToken
    uint public totalRewardToken;
```

```
// The account used to manage the rewards which are to be distributed to the
// DAO Token Holders of any DAO that holds Reward Tokens
ManagedAccount public rewardAccount;
// Amount of rewards (in wei) already paid out to a certain address
mapping (address => uint) public paidOut;
// Map of addresses blocked during a vote (not allowed to transfer DAO
// tokens). The address points to the proposal ID.
mapping (address => uint) public blocked;

// The minimum deposit (in wei) required to submit any proposal that is not
// requesting a new service provider (no deposit is required for splits)
uint public proposalDeposit;

// Contract that is able to create a new DAO (with the same code as
// this one), used for splits
DAO_Creator public daoCreator;

// A proposal with `newServiceProvider == false` represents a transaction
// to be issued by this DAO
// A proposal with `newServiceProvider == true` represents a DAO split
struct Proposal {
    // The address where the `amount` will go to if the proposal is accepted
    // or if `newServiceProvider` is true, the proposed service provider of
    //the new DAO).
    address recipient;
    // The amount to transfer to `recipient` if the proposal is accepted.
    uint amount;
    // A plain text description of the proposal
    string description;
    // A unix timestamp, denoting the end of the voting period
    uint votingDeadline;
    // True if the proposal's votes have yet to be counted, otherwise False
    bool open;
    // True if quorum has been reached, the votes have been counted, and
    // the majority said yes
    bool proposalPassed;
    // A hash to check validity of a proposal
    bytes32 proposalHash;
    // Deposit in wei the creator added when submitting their proposal. It
    // is taken from the msg.value of a newProposal call.
    uint proposalDeposit;
    // True if this proposal is to assign a new service provider
    bool newServiceProvider;
    // Data needed for splitting the DAO
    SplitData[] splitData;
    // Number of tokens in favour of the proposal
    uint yea;
    // Number of tokens opposed to the proposal
    uint nay;
    // Simple mapping to check if a shareholder has voted for it
    mapping (address => bool) votedYes;
    // Simple mapping to check if a shareholder has voted against it
    mapping (address => bool) votedNo;
    // Address of the shareholder who created the proposal
    address creator;
}

// Used only in the case of a newServiceProvider porposal.
struct SplitData {
    // The balance of the current DAO minus the deposit at the time of split
    uint splitBalance;
    // The total amount of DAO Tokens in existence at the time of split.
    uint totalSupply;
```

```
    // Amount of Reward Tokens owned by the DAO at the time of split.
    uint rewardToken;
    // The new DAO contract created at the time of split.
    DAO newDAO;
}
// Used to restrict acces to certain functions to only DAO Token Holders
modifier onlyTokenholders {}

/// @dev Constructor setting the default service provider and the address
/// for the contract able to create another DAO as well as the parameters
/// for the DAO Token Sale
/// @param _defaultServiceProvider The default service provider
/// @param _daoCreator The contract able to (re)create this DAO
/// @param _minValue Minimal value for a successful DAO Token Sale
/// @param _closingTime Date (in unix time) of the end of the DAO Token Sale
/// @param _privateSale If zero the DAO Token Sale is open to public, a
/// non-zero address means that the DAO Token Sale is only for the address
// This is the constructor: it can not be overloaded so it is commented out
//  function DAO(
    //  address _defaultServiceProvider,
    //  DAO_Creator _daoCreator,
    //  uint _minValue,
    //  uint _closingTime,
    //  address _privateSale
//  )

/// @notice Buy Token with `msg.sender` as the beneficiary
function () returns (bool success);

/// @dev Function used by the products of the DAO (e.g. Slocks) to send
/// rewards to the DAO
/// @return Whether the call to this function was successful or not
function payDAO() returns(bool);

/// @dev This function is used by the service provider to send money back
/// to the DAO, it can also be used to receive payments that should not be
/// counted as rewards (donations, grants, etc.)
/// @return Whether the DAO received the ether successfully
function receiveEther() returns(bool);

/// @notice `msg.sender` creates a proposal to send `_amount` Wei to
///  `_recipient` with the transaction data `_transactionData`. If
/// `_newServiceProvider` is true, then this is a proposal that splits the
/// DAO and sets `_recipient` as the new DAO's new service provider.
/// @param _recipient Address of the recipient of the proposed transaction
/// @param _amount Amount of wei to be sent with the proposed transaction
/// @param _description String describing the proposal
/// @param _transactionData Data of the proposed transaction
/// @param _debatingPeriod Time used for debating a proposal, at least 2
/// weeks for a regular proposal, 10 days for new service provider proposal
/// @param _newServiceProvider Bool defining whether this proposal is about
/// a new service provider or not
/// @return The proposal ID. Needed for voting on the proposal
function newProposal(
    address _recipient,
    uint _amount,
    string _description,
    bytes _transactionData,
    uint _debatingPeriod,
    bool _newServiceProvider
) onlyTokenholders returns (uint _proposalID);

/// @notice Check that the proposal with the ID `_proposalID` matches the
```

```
/// transaction which sends ‘_amount‘ with data ‘_transactionData‘
/// to ‘_recipient‘
/// @param _proposalID The proposal ID
/// @param _recipient The recipient of the proposed transaction
/// @param _amount The amount of wei to be sent in the proposed transaction
/// @param _transactionData The data of the proposed transaction
/// @return Whether the proposal ID matches the transaction data or not
function checkProposalCode(
    uint _proposalID,
    address _recipient,
    uint _amount,
    bytes _transactionData
) constant returns (bool _codeChecksOut);

/// @notice Vote on proposal ‘_proposalID‘ with ‘_supportsProposal‘
/// @param _proposalID The proposal ID
/// @param _supportsProposal Yes/No - support of the proposal
/// @return The vote ID.
function vote(
    uint _proposalID,
    bool _supportsProposal
) onlyTokenholders returns (uint _voteID);

/// @notice Checks whether proposal ‘_proposalID‘ with transaction data
/// ‘_transactionData‘ has been voted for or rejected, and executes the
/// transaction in the case it has been voted for.
/// @param _proposalID The proposal ID
/// @param _transactionData The data of the proposed transaction
/// @return Whether the proposed transaction has been executed or not
function executeProposal(
    uint _proposalID,
    bytes _transactionData
) returns (bool _success);

/// @notice ATTENTION! I confirm to move my remaining funds to a new DAO
/// with ‘_newServiceProvider‘ as the new service provider, as has been
/// proposed in proposal ‘_proposalID‘. This will burn my tokens. This can
/// not be undone and will split the DAO into two DAO’s, with two
/// different underlying tokens.
/// @param _proposalID The proposal ID
/// @param _newServiceProvider The new service provider of the new DAO
/// @dev This function, when called for the first time for this proposal,
/// will create a new DAO and send the sender’s portion of the remaining
/// ether and Reward Tokens to the new DAO. It will also burn the DAO Tokens
/// of the sender.
function splitDAO(
    uint _proposalID,
    address _newServiceProvider
) returns (bool _success);

/// @notice Add a new possible recipient ‘_recipient‘ to the whitelist so
/// that the DAO can send transactions to them (using proposals)
/// @param _recipient New recipient address
/// @dev Can only be called by the current service provider
function addAllowedAddress(address _recipient) external returns (bool _success);

/// @notice Change the minimum deposit required to submit a proposal
/// @param _proposalDeposit The new proposal deposit
/// @dev Can only be called by this DAO (through proposals with the
/// recipient being this DAO itself)
function changeProposalDeposit(uint _proposalDeposit) external;

/// @notice Get my portion of the reward that was sent to ‘rewardAccount‘
```

```
    /// @return Whether the call was successful
    function getMyReward() returns(bool _success);

    /// @notice Withdraw `account`'s portion of the reward from `rewardAccount`,
    /// to `account`'s balance
    /// @return Whether the call was successful
    function withdrawRewardFor(address _account) returns(bool _success);

    /// @notice Send `_amount` tokens to `_to` from `msg.sender`. Prior to this
    /// getMyReward() is called.
    /// @param _to The address of the recipient
    /// @param _amount The amount of tokens to be transfered
    /// @return Whether the transfer was successful or not
    function transferWithoutReward(address _to, uint256 _amount) returns (bool success);

    /// @notice Send `_amount` tokens to `_to` from `_from` on the condition it
    /// is approved by `_from`. Prior to this getMyReward() is called.
    /// @param _from The address of the sender
    /// @param _to The address of the recipient
    /// @param _amount The amount of tokens to be transfered
    /// @return Whether the transfer was successful or not
    function transferFromWithoutReward(
        address _from,
        address _to,
        uint256 _amount
    ) returns (bool success);

    /// @notice Doubles the 'minQuorumDivisor' in the case quorum has not been
    /// achieved in 52 weeks
    /// @return Whether the change was successful or not
    function halveMinQuorum() returns (bool _success);

    /// @return total number of proposals ever created
    function numberOfProposals() constant returns (uint _numberOfProposals);

    /// @param _account The address of the account which is checked.
    /// @return Whether the account is blocked (not allowed to transfer tokens) or not.
    function isBlocked(address _account) returns (bool);


    event ProposalAdded(
        uint indexed proposalID,
        address recipient,
        uint amount,
        bool newServiceProvider,
        string description
    );
    event Voted(uint indexed proposalID, bool position, address indexed voter);
    event ProposalTallied(uint indexed proposalID, bool result, uint quorum);
    event NewServiceProvider(address indexed _newServiceProvider);
    event AllowedRecipientAdded(address indexed _recipient);
}

// The DAO contract itself
contract DAO is DAOInterface, Token, TokenSale {

    // Modifier that allows only shareholders to vote and create new proposals
    modifier onlyTokenholders {
        if (balanceOf(msg.sender) == 0) throw;
            _
    }

    function DAO(
```

```
    address _defaultServiceProvider,
    DAO_Creator _daoCreator,
    uint _minValue,
    uint _closingTime,
    address _privateSale
) TokenSale(_minValue, _closingTime, _privateSale) {

    serviceProvider = _defaultServiceProvider;
    daoCreator = _daoCreator;
    proposalDeposit = 20 ether;
    rewardAccount = new ManagedAccount(address(this));
    lastTimeMinQuorumMet = now;
    minQuorumDivisor = 5; // sets the minimal quorum to 20%
    proposals.length++; // avoids a proposal with ID 0 because it is used
    if (address(rewardAccount) == 0)
        throw;
}

function () returns (bool success) {
    if (now < closingTime + 40 days)
        return buyTokenProxy(msg.sender);
    else
        return receiveEther();
}


function payDAO() returns (bool) {
    rewards += msg.value;
    return true;
}

function receiveEther() returns (bool) {
    return true;
}


function newProposal(
    address _recipient,
    uint _amount,
    string _description,
    bytes _transactionData,
    uint _debatingPeriod,
    bool _newServiceProvider
) onlyTokenholders returns (uint _proposalID) {

    // Sanity check
    if (_newServiceProvider && (
        _amount != 0
        || _transactionData.length != 0
        || _recipient == serviceProvider
        || msg.value > 0
        || _debatingPeriod < 1 weeks)) {
        throw;
    } else if(
        !_newServiceProvider
        && (!isRecipientAllowed(_recipient) || (_debatingPeriod < 2 weeks))
    ) {
        throw;
    }

    if (!isFunded
        || now < closingTime
        || (msg.value < proposalDeposit && !_newServiceProvider)) {
```

```
        throw;
    }

    if (_recipient == address(rewardAccount) && _amount > rewards)
        throw;

    if (now + _debatingPeriod < now) // prevents overflow
        throw;

    _proposalID = proposals.length++;
    Proposal p = proposals[_proposalID];
    p.recipient = _recipient;
    p.amount = _amount;
    p.description = _description;
    p.proposalHash = sha3(_recipient, _amount, _transactionData);
    p.votingDeadline = now + _debatingPeriod;
    p.open = true;
    //p.proposalPassed = False; // that's default
    p.newServiceProvider = _newServiceProvider;
    if (_newServiceProvider)
        p.splitData.length++;
    p.creator = msg.sender;
    p.proposalDeposit = msg.value;
    ProposalAdded(
        _proposalID,
        _recipient,
        _amount,
        _newServiceProvider,
        _description
    );
}


function checkProposalCode(
    uint _proposalID,
    address _recipient,
    uint _amount,
    bytes _transactionData
) noEther constant returns (bool _codeChecksOut) {
    Proposal p = proposals[_proposalID];
    return p.proposalHash == sha3(_recipient, _amount, _transactionData);
}


function vote(
    uint _proposalID,
    bool _supportsProposal
) onlyTokenholders noEther returns (uint _voteID) {

    Proposal p = proposals[_proposalID];
    if (p.votedYes[msg.sender]
        || p.votedNo[msg.sender]
        || now >= p.votingDeadline) {

        throw;
    }

    if (_supportsProposal) {
        p.yea += balances[msg.sender];
        p.votedYes[msg.sender] = true;
    } else {
        p.nay += balances[msg.sender];
```

```
        p.votedNo[msg.sender] = true;
    }

    if (blocked[msg.sender] == 0) {
        blocked[msg.sender] = _proposalID;
    } else if (p.votingDeadline > proposals[blocked[msg.sender]].votingDeadline) {
        // this proposal's voting deadline is further into the future than
        // the proposal that blocks the sender so make it the blocker
        blocked[msg.sender] = _proposalID;
    }

    Voted(_proposalID, _supportsProposal, msg.sender);
}


function executeProposal(
    uint _proposalID,
    bytes _transactionData
) noEther returns (bool _success) {

    Proposal p = proposals[_proposalID];
    // Check if the proposal can be executed
    if (now < p.votingDeadline  // has the voting deadline arrived?
        // Have the votes been counted?
        || !p.open
        // Does the transaction code match the proposal?
        || p.proposalHash != sha3(p.recipient, p.amount, _transactionData)) {

        throw;
    }

    if (p.newServiceProvider) {
        p.open = false;
        return;
    }

    uint quorum = p.yea + p.nay;

    // Execute result
    if (quorum >= minQuorum(p.amount) && p.yea > p.nay) {
        if (!p.creator.send(p.proposalDeposit))
            throw;
        // Without this throw, the creator of the proposal can repeat this,
        // and get so much ether
        if (!p.recipient.call.value(p.amount)(_transactionData))
            throw;
        p.proposalPassed = true;
        _success = true;
        lastTimeMinQuorumMet = now;
        if (p.recipient == address(rewardAccount)) {
            // This happens when multiple similar proposals are created and
            // both are passed at the same time.
            if (rewards < p.amount)
                throw;
            rewards -= p.amount;
        } else {
            rewardToken[address(this)] += p.amount;
            totalRewardToken += p.amount;
        }
    } else if (quorum >= minQuorum(p.amount) && p.nay >= p.yea) {
        if (!p.creator.send(p.proposalDeposit))
            throw;
        lastTimeMinQuorumMet = now;
```

```
    }

    // Since the voting deadline is over, close the proposal
    p.open = false;

    // Initiate event
    ProposalTallied(_proposalID, _success, quorum);
}


function splitDAO(
    uint _proposalID,
    address _newServiceProvider
) noEther onlyTokenholders returns (bool _success) {

    Proposal p = proposals[_proposalID];

    // Sanity check

    if (now < p.votingDeadline  // has the voting deadline arrived?
        //The request for a split expires 41 days after the voting deadline
        || now > p.votingDeadline + 41 days
        // Does the new service provider address match?
        || p.recipient != _newServiceProvider
        // Is it a new service provider proposal?
        || !p.newServiceProvider
        // Have you voted for this split?
        || !p.votedYes[msg.sender]
        // Did you already vote on another proposal?
        || blocked[msg.sender] != _proposalID) {

        throw;
    }

    // If the new DAO doesn't exist yet, create the new DAO and store the
    // current split data
    if (address(p.splitData[0].newDAO) == 0) {
        p.splitData[0].newDAO = createNewDAO(_newServiceProvider);
        // Call depth limit reached, etc.
        if (address(p.splitData[0].newDAO) == 0)
            throw;
        // p.proposalDeposit should be zero here
        if (this.balance < p.proposalDeposit)
            throw;
        p.splitData[0].splitBalance = this.balance - p.proposalDeposit;
        p.splitData[0].rewardToken = rewardToken[address(this)];
        p.splitData[0].totalSupply = totalSupply;
        p.proposalPassed = true;
    }

    // Move funds and assign new Tokens
    uint fundsToBeMoved =
        (balances[msg.sender] * p.splitData[0].splitBalance) /
        p.splitData[0].totalSupply;
    if (p.splitData[0].newDAO.buyTokenProxy.value(fundsToBeMoved)(msg.sender) == false)
        throw;


    // Assign reward rights to new DAO
    uint rewardTokenToBeMoved =
        (balances[msg.sender] * p.splitData[0].rewardToken) /
        p.splitData[0].totalSupply;
    rewardToken[address(p.splitData[0].newDAO)] += rewardTokenToBeMoved;
```

```
    if (rewardToken[address(this)] < rewardTokenToBeMoved)
        throw;
    rewardToken[address(this)] -= rewardTokenToBeMoved;

    // Burn DAO Tokens
    Transfer(msg.sender, 0, balances[msg.sender]);
    totalSupply -= balances[msg.sender];
    balances[msg.sender] = 0;
    paidOut[address(p.splitData[0].newDAO)] += paidOut[msg.sender];
    paidOut[msg.sender] = 0;

    return true;
}


function getMyReward() noEther returns (bool _success) {
    return withdrawRewardFor(msg.sender);
}


function withdrawRewardFor(address _account) noEther returns (bool _success) {
    // The account's portion of Reward Tokens of this DAO
    uint portionOfTheReward =
        (balanceOf(_account) * rewardToken[address(this)]) /
        totalSupply + rewardToken[_account];
    uint reward =
        (portionOfTheReward * rewardAccount.accumulatedInput()) /
        totalRewardToken - paidOut[_account];
    if (!rewardAccount.payOut(_account, reward))
        throw;
    paidOut[_account] += reward;
    return true;
}


function transfer(address _to, uint256 _value) returns (bool success) {
    if (isFunded
        && now > closingTime
        && !isBlocked(msg.sender)
        && transferPaidOut(msg.sender, _to, _value)
        && super.transfer(_to, _value)) {

        return true;
    } else {
        throw;
    }
}


function transferWithoutReward(address _to, uint256 _value) returns (bool success) {
    if (!getMyReward())
        throw;
    return transfer(_to, _value);
}


function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {
    if (isFunded
        && now > closingTime
        && !isBlocked(_from)
        && transferPaidOut(_from, _to, _value)
        && super.transferFrom(_from, _to, _value)) {
```

```
        return true;
    } else {
        throw;
    }
}


function transferFromWithoutReward(
    address _from,
    address _to,
    uint256 _value
) returns (bool success) {

    if (!withdrawRewardFor(_from))
        throw;
    return transferFrom(_from, _to, _value);
}


function transferPaidOut(
    address _from,
    address _to,
    uint256 _value
) internal returns (bool success) {

    uint transferPaidOut = paidOut[_from] * _value / balanceOf(_from);
    if (transferPaidOut > paidOut[_from])
        throw;
    paidOut[_from] -= transferPaidOut;
    paidOut[_to] += transferPaidOut;
    return true;
}


function changeProposalDeposit(uint _proposalDeposit) noEther external {
    if (msg.sender != address(this) || _proposalDeposit > this.balance / 10)
        throw;
    proposalDeposit = _proposalDeposit;
}


function addAllowedAddress(address _recipient) noEther external returns (bool _success) {
    if (msg.sender != serviceProvider)
        throw;
    allowedRecipients.push(_recipient);
    return true;
}


function isRecipientAllowed(address _recipient) internal returns (bool _isAllowed) {
    if (_recipient == serviceProvider
        || _recipient == address(rewardAccount)
        || _recipient == address(this)
        || (_recipient == address(extraBalance)
            // only allowed when at least the amount held in the
            // extraBalance account has been spent from the DAO
            && totalRewardToken > extraBalance.accumulatedInput()))
        return true;

    for (uint i = 0; i < allowedRecipients.length; ++i) {
        if (_recipient == allowedRecipients[i])
            return true;
    }
```

```
        return false;
    }


    function minQuorum(uint _value) internal returns (uint _minQuorum) {
        // minimum of 20% and maximum of 53.33%
        return totalSupply / minQuorumDivisor + _value / 3;
    }


    function halveMinQuorum() returns (bool _success) {
        if (lastTimeMinQuorumMet < (now - 52 weeks)) {
            lastTimeMinQuorumMet = now;
            minQuorumDivisor *= 2;
            return true;
        } else {
            return false;
        }
    }


    function createNewDAO(address _newServiceProvider) internal returns (DAO _newDAO) {
        NewServiceProvider(_newServiceProvider);
        return daoCreator.createDAO(_newServiceProvider, 0, now + 42 days);
    }


    function numberOfProposals() constant returns (uint _numberOfProposals) {
        // Don't count index 0. It's used by isBlocked() and exists from start
        return proposals.length - 1;
    }


    function isBlocked(address _account) returns (bool) {
        if (blocked[_account] == 0)
            return false;
        Proposal p = proposals[blocked[_account]];
        if (!p.open) {
            blocked[_account] = 0;
            return false;
        } else {
            return true;
        }
    }
}

contract DAO_Creator {
    function createDAO(
        address _defaultServiceProvider,
        uint _minValue,
        uint _closingTime
    ) returns (DAO _newDAO) {

        return new DAO(
            _defaultServiceProvider,
            DAO_Creator(this),
            _minValue,
            _closingTime,
            msg.sender
        );
    }
}
```

A.4. **Sample Offer.**

```
contract SampleOffer {

    uint totalCosts;
    uint oneTimeCosts;
    uint dailyCosts;


    address serviceProvider;
    bytes32 hashOfTheContract;
    uint minDailyCosts;
    uint paidOut;

    uint dateOfSignature;
    DAO client; // address of DAO

    bool public promiseValid;
    uint public rewardDivisor;
    uint public deploymentReward;

    modifier callingRestriction {
        if (promiseValid) {
            if (msg.sender != address(client))
                throw;
        } else if (msg.sender != serviceProvider) {
                throw;
        }
        _
    }

    modifier onlyClient {
        if (msg.sender != address(client))
            throw;
        _
    }

    function SampleOffer(
        address _serviceProvider,
        bytes32 _hashOfTheContract,
        uint _totalCosts,
        uint _oneTimeCosts,
        uint _minDailyCosts,
        uint _rewardDivisor,
        uint _deploymentReward
    ) {
        serviceProvider = _serviceProvider;
        hashOfTheContract = _hashOfTheContract;
        totalCosts = _totalCosts;
        oneTimeCosts = _oneTimeCosts;
        minDailyCosts = _minDailyCosts;
        dailyCosts = _minDailyCosts;
        rewardDivisor = _rewardDivisor;
        deploymentReward = _deploymentReward;
    }

    function sign() {
        if (msg.value < totalCosts && dateOfSignature != 0)
            throw;
        if (!serviceProvider.send(oneTimeCosts))
            throw;
        client = DAO(msg.sender);
        dateOfSignature = now;
        promiseValid = true;
```

```
}

function setDailyCosts(uint _dailyCosts) onlyClient {
    dailyCosts = _dailyCosts;
    if (dailyCosts < minDailyCosts)
        promiseValid = false;
}

function returnRemainingMoney() onlyClient {
    if (client.receiveEther.value(this.balance)())
        promiseValid = false;
}

function getDailyPayment() {
    if (msg.sender != serviceProvider)
        throw;
    uint amount = (now - dateOfSignature) / (1 days) * dailyCosts - paidOut;
    if (serviceProvider.send(amount))
        paidOut += amount;
}

function setRewardDivisor(uint _rewardDivisor) callingRestriction {
    if (_rewardDivisor < 50 && msg.sender != address(client))
        throw; // 2% is the default max reward
    rewardDivisor = _rewardDivisor;
}

function setDeploymentFee(uint _deploymentReward) callingRestriction {
    if (deploymentReward > 10 ether && msg.sender != address(client))
        throw;
    deploymentReward = _deploymentReward;
}

// interface for Slocks
function payOneTimeReward() returns(bool) {
    if (msg.value < deploymentReward)
        throw;
    if (promiseValid) {
        if (client.payDAO.value(msg.value)()) {
            return true;
        } else {
            throw;
        }
    } else {
        if (serviceProvider.send(msg.value)) {
            return true;
        } else {
            throw;
        }
    }
}

// pay reward
function payReward() returns(bool) {
    if (promiseValid) {
        if (client.payDAO.value(msg.value)()) {
            return true;
        } else {
            throw;
        }
    } else {
        if (serviceProvider.send(msg.value)) {
            return true;
```

```
            } else {
                throw;
            }
        }
    }
}
```

A.5. **Managed Account.**

```
contract ManagedAccountInterface {
    address public owner;
    uint public accumulatedInput;

    function payOut(address _recipient, uint _amount) returns (bool);

    event PayOut(address indexed _recipient, uint _amount);
}


contract ManagedAccount is ManagedAccountInterface{
    function ManagedAccount(address _owner){
        owner = _owner;
    }

    function(){
        accumulatedInput += msg.value;
    }

    function payOut(address _recipient, uint _amount) returns (bool){
        if (msg.sender != owner || msg.value > 0) throw;
        if (_recipient.call.value(_amount)()){
            PayOut(_recipient, _amount);
            return true;
        }
        else
            return false;
    }
}
```