摘要

该文档描述了第一个去中心化的管理信任的DAO实现。该概念，论述了大多数劫持少数攻击路径的问题，并提出一种解决方案。并且，一种针对这种类型的DAO的切实可行的方案，提出并进行详细描述，它是使用Solidity运行在以太坊上的智能合约。

去中心化的自治组织来管理信任
最终草稿 - 审稿中

Christoph Jentzsch
Founder & CTO, Slock.it
christoph.jentzsch@slock.it
译者: 张亚宁 - ethfans.org

2016年5月1日

1 序言

历史上，信用的管理始终是中心化的。一群个体（受托人）会制定一套合约并结合他们的资源形成信任，然后由一个人或者一个公司来管理，受托人失去了直接管理他们资产的直接权利。众筹(Massolution [2015])的崛起降低了小的投资人参与大项目的门槛。众筹的资金仍然带有很高的风险，由于公司管理不善经常会未能兑现承诺或者干脆直接消失的无影无踪(Knibbs [2015], Biggs [2015])。谢谢以太坊的开发者们(Buterin [2013], Wood [2014])，它集成图灵完备语言和运行智能合约的能力，使的创建一个受托人直接管理他们资金的成为可能，同时仍然能保证众多小的受托人能一起实现同一个大的目标。我们已经在区块链上使用智能合约(Szabo [1997], Miller [1997])来形成一个DAO组织。在这个白皮书里，我们会通过一个具体的例子来详细介绍DAO管理信任的概念。

在解释了DAO概念之后，我们会讨论“多数抢劫少数攻击“并提出一种解决方案，“DAO分离“。最后我们会浏览智能合约，以更细节的“DAO分离“来结束。

智能合约的地址在：`https://github.com/slockit/DAO/`。

2 概念

这一节会介绍DAO的基本概念和它能够做什么。

在DAO部署到以太坊区块链之后，在规定的起始筹资阶段，任何人都可以通过向DAO智能合约地址发送以太币（支持以太坊网络的数字燃料）的方式来参与众筹。作为交换，代币会被创建用来代表是会员身份以及DAO一部分的所有权;这些代币会被分配给众筹的参与者。代币的数量是按照发送以太币的比例分配的。每一个代币的价格会随着时间而变化（见章节 5）。在众筹结束后，这些代币的所有权可以通过以太坊区块链的交易转移给其他用户。在部署合约时，一个最小的DAO众筹目标和起始众筹时间会被设定。如果最小众筹金额在众筹阶段没有被满足，每一个众筹者的以太币会被返还。在众筹阶段结束后，我们用Ξ_{raised}来表示众筹来的总资金，用T_{total}来表示创建的总的代币数量。这个DAO仅仅用来管理众筹金额。它本身没有生产产品，编写代码和开发硬件的能力。它需要一个服务提供商来完成这些和其他目标，通过签署提议的方式来临时租用他们。DAO的每一个成员都可以花费一部分众筹来的以太币来提交提议，在项目中记作Ξ_{transfer}。如果建议被批准，以太币会发送到另外一个表示提议项目的合约中。这样，智能合约可以参数化，使DAO能与资助的项目相

互作用和影响它。一个在DAO和资助项目之间的协议的例子可以在附录中找到(A.4)。

DAO的成员投票权重由他们掌握的代币数量来决定。代币是可分割，无差别的，可以方便的在用户之间进行转移。在合约中，成员的个人行为不能被直接确定。任何提议都需要一个时间范围t_p去讨论和投票。在我们的例子中，这个时间范围是由提议的创建者设定的，对于一般的提议最少需要两周的时间。

在t_p时间后，代币的持有者会调用一个在DAO合约中的函数来验证大多数的投票是支持提议的并达到了法定人数。如果是这种情况，提议将会执行。如果不是这种情况，提议将会关闭。最小的法定人数表示投票成立的最小代币数，标记为q_{min}，计算方式如下：

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

d是`minQuorumDivisor`。这个参数默认值是5，如果法定人数在超过一年仍未满足，它会加倍。

Ξ_{DAO}是DAO拥有的总的以太币数量，R_{total}是总的奖励代币，在7 节会提到（同样指`totalRewardToken` 在A.3）。 $\Xi_{DAO} + R_{\text{total}}$ 的和等于总共众筹得来的以太币加上收到的奖励。

这意味着，所有的提议如果要通过，起始的20%的所有的代币的法定数字是必须的。如果Ξ_{transfer}和众筹的以太币加上收到的奖励相等，至少53.33%的法定数字是必须的。

为了组织提议的垃圾化，新建提议时需要支付最小押金，如果法定数达到将返还押金，如果不足提议的押金将保留在DAO中。DAO里默认的提议押金值可以通过一个其他提议来修改。

3 记法

在本文中，Ξ 总是表示以wei为单位的以太币。定义为1 Wei = 10^{-18} Ether (Wood [2014])。

同样地，DAO代币标记为T，总是代表以它基本单位为单位DAO代币。定义为10^{-16} DAO代币。

4 大多数抢劫少数攻击

每一个DAO都需要去减缓的问题是，大多数抢劫少数。如果攻击者有51%的代币，在众筹期间获得或者后面购买的方式，都可以创建一个提议发送所有的金额给他们。因为他们拥有绝大多数的token，所以他们总是能通过提议。

为了防止这种现象，少数人总是能选择取回他们众筹的部分金额。可以通过将DAO一分为二来实现。在这种情况下，个人或者一群代币的所有者，非常反对某项提议，在某项提议执行前想要取回他们的资金，他们可以提交一个特殊的提议来形成一个新的DAO。少数人可以投票将他们的资金转移到这个新的DAO中，使的剩下的大多数只能花费他们自己的钱。这个想法起源于Vitalik Buterin (Buterin [2015])发表的一篇博客。

这种简单的改进存在的问题是，它不能处理投票冷淡：一些代币的拥有者或许并不会积极的参与到DAO中，不能紧密的跟进提议。攻击者可以利用这点作为优势。即使少数人有机会取回他们的资金并且分离DAO，但是他们其中的一些人可能不能及时意识到情况而不会这样做。为了DAO安全，它需要考虑那些不活跃的代币拥有者不会丢失他们的资金。我们建议的解决方法是，限制每个单独的DAO对应一个单独的服务提供商。这个服务提供方控制着一个唯一的账号，通过提议可以从DAO中接受。另外，服务器提供商可以创建DAO可以发送金额的白名单地址。这给了服务提供方非常大的权力。为了防止滥用这种权力，DAO可以投票选择新的服务提供方，或许有可能导致前面描述的DAO一分为二的情况发生。

任何一个代币的持有者可以提交一个提议去选举新的服务提供方。实际上，即使一个单独的token拥有者也能够取回他们剩余的ether份额和维持他们在未来的应得的收益（根据之前的贡献决定），这些都会自动发送到新的DAO中。收益的定义是，从DAO众筹起，DAO从产品生产中获得的任何ether，这将会在第7节中进一步详细阐述。

选举新的服务提供方过程如下：任何一个token持有者可以发起一个提议来选举新的服务提供方。这个提议需要押金支付，否则攻击者可以

投票修改为一个非常高的押金，来阻止任何分离。这个提议的投票期为10天。

这比常规提议的最低期限要少4天，是为了允许任何人取回他们的资金，在任何潜在的恶意提议通过之前。同时没有法定额度限制，所以任何token持有者有权力分离去他们自己的DAO。

讨论期通常会讨论新的服务提供方，进行非正式的投票。投票的结果不会产生任何实际作用，它只有纯粹的指导性功能。在第一轮投票之后，代币的持有者可以进行第二轮的投票来确认结果。大多数的人可以投票保持原有的服务提供方来避免分离，或者相反他们可以投票新的服务提供方来将他们的资金份额转移到新的DAO中。

5 代币价格

为了奖励在众筹阶段购买代币的参与者，因为他们有更少的信息，所以相对于后加入的人承受了更大的风险，他们会支付更少相对于后加入的人。

就现在描述的DAO而言，我们选择了以下的价格计算方式：

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

乘数m的定义如下：

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

这里的t是unit秒时间，t_c是众筹的关闭时间（见A.2 的`closingTime`），w是一周的秒数，d是一天的秒数。

因此每一个购买者获取的代币的计算方式如下：$P(t) \cdot \Xi_c$。这里Ξ_c表示支付的以太币，单位是wei。

这个结果是一个常量价格在开始的时候，至到2周后代币出售结束的时间。在这个时间，每个DAO代币的价格每天会按照0.005 Ξ_c 的速度来增长。至到众筹结束前的第四天，每个DAO代币会是一个固定的价格 0.015 Ξ_c。

价格的增长会导致一种情况，一个单独的参与者在初试价格购买了代币后，在预售结束后立即分离出一个DAO后悔得到更多的以太币，因为其他的参与者支付了更高的价格(Green [2016])。

为了避免这种可能性，所有比购买初始价格高的代币的以太币，会被发送到一个额外的帐户。在A.2标记为`extraBalance`。这个钱可以被发回到DAO中，通过一个提议，在DAO已经花费至少这个金额的钱之后。

这个规则已经实现为一个内部函数`isRecipientAllowed`在 6.3 节。

6 合约

本节主要详述智能合约实现上述的概念。合约使用Solidity(Reitwiessner and Wood [2015])编写。

每一个合约包含成员变量和函数，这样外部就可以通过向以太坊网络发送交易的方式来访问，以DAO合约地址为接受者，方法ID（参数可选）为数据。在这一节我们会详细讨论变量和函数的含义。

主合约被称作'DAO' 。它定义了DAO的内部工作方式，它的成员变量和函数来源于'Token'和'TokenSale'。 'Token'定义了DAO代币的内部工作方式， 'TokenSale'定义如何使用如何使用以太币购买DAO代币。除了这三个合约外，还有'ManagedAccount'合约，它作为一个辅助合约用来保存分发给代币持有者的奖励，还有'extraBalance'合约 （见第5节）。合约'SampleOffer'(A.4)是一个从服务提供方到DAO的提议的范例。

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);
}
```

以上是'Token'合约的接口。这些合约的接口起到文档的作用，可以概述合约中的函数和变量。全部的实现可以在附录中找到(A.1)。这个合约展示了一种标准代币: https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs， 合约https://github.com/ConsenSys/Tokens/blob/master/Token_Contracts/contracts/Standard_Token.sol 是其他合约创建的基础。

map类型的balances存储了DAO成员的代币， 以address做索引。所有继承实现了 TokenInterface的合约， 都可以直接修改map的内容，但是只有4个方法可以这样做: buyTokenProxy, transfer , transferFrom 和splitDAO。

map类型的allowed用做记录预定的地址，这些地址允许以其他人的名义发送代币。

integer类型的totalSupply是现存的DAO代币的总数量。public 关键字创建一个同名函数，它用来返回变量的值，所以被称为公开变量。

函数balanceOf返回特定地址的余额。

函数transfer用作请求者发送代币给其他地址。

函数transferFrom用作代表某人来发送以太币，并且先前已经使用approve 函数批准。

函数approve用做DAO代币的所有者，指定一个特定的spender来转移指定 value 的金额从他们帐户，使用transferFrom函数。如果想检测某个地址是否被允许代表某人使用DAO代币，可以使用函数allowance，它会返回允许spender是否可以花费的代币。这有点类似于写支票。

事件Transfer 用来通知轻客户端 balances的变化。

事件 Approval 用来通知轻客户端allowed的变化。

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);
}
```

以上是TokenSale合约(A.2)的接口。

integer类型的closingTime是代币预售期结束的(unix)时间戳。

integer类型的minValue是DAO众筹时需要接受的值，单位是wei。

boolean类型的isFunded是ture如果DAO已经满足最低众筹目标，否则false。

地址类型privateSale用作DAO的分离 - 如果设置为0，表示为公开出售，否则只有存储在privateSale里的地址才能购买代币。

管理帐号(A.5)extraBalance用来保存，在众筹期间，代币价格上涨之后的多出的以太币。任何以高于初始价格的支付的以太币会到这个账户。

map类型的weiGiven用来保存每一个在众筹期间的参与者的众筹金额，它只有一个用途，即如果代币预售没有到达募集目标时，将以太币返回给参与者。

结构体TokenSale初始化了代币预售期使用的参数，包括minValue, closingtime，privateSale，这些值会在DAO合约(A.3)中设置，且只会在DAO部署的时候运行一次。

函数buyTokenProxy为每个wei的发送，创建了一个DAO代币的最小面额。价格计算方式为：

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

这里Ξ_c是以wei单位的金额用以购买代币，除数divisor的大小取决于时间，这在5节中有介绍。

参数tokenHolder定义了新挖到的代币的接受者。

函数refund可以被任何一个参与者调用，如果预售失败未满足众筹目标，它会讲参与者的以太币返还。

函数divisor用以计算，函数buyTokenProxy在预售期间，代币的价格。

事件FundingToDate, SoldToken和Refund用来通知轻客户端众筹的状态。

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);
}
```

原始合约是以 ‘’ DAO ‘’: http://chriseth.github.io/browser-solidity/?gist=192371538cf5e43e6dab） 为基础，在https://blog.ethereum.org/2015/12/04有描述。主要的增加了分离机制和一些跟它相关的内容。现在我们可以定义将成员变量和函数定义一次。

数组proposals保存了所有的提议。

整型minQuorumDivisor用于计算需要提议通过的法定数。它被设置为5，如果法定数超过一年没有达到，这个数字会加倍。

整型lastTimeMinQuorumMet保存了法定数满足的最后时间变化。

整型rewards计算所有发送给DAO的奖励。在支付到rewardAccount之后，它会被重置为0.

地址类型serviceProvider，由DAO的创建者设置，它定义了服务提供方。

列表类型的allowedRecipients一般被当做白名单。DAO只能发送交易给它自己serviceProvider, rewardAccount, extraBalance以及白名单里的地址。只有serviceProvider（服务提供方）能向白名单里增加地址。

map类型的rewardToken记录了，由DAO产品产生的奖励所在的地址。这些地址只能DAO的地址。

整型totalRewardToken记录的现存的奖励代币数量。

变量rewardAccount是ManagedAccount类型的，在A.5中讨论过。它用来管理分发给DAO代币持有者的奖励以及发送奖励给代币持有者。

map类型的paidOut用于记录一个代币持有者已经取回多少wei从rewardAccount中。

map类型的blocked用来保存DAO 代币里已经参与投票的地址，这样只有在投票完成后，这些钱才能被转让。这些地址指向提议的ID。

整型proposalDeposit指定了任何提议需要支付的最小押金，但是不包括变更服务提供方。

合约daoCreator用于创建一个新的DAO和这个DAO同样的代码，用于DAO分离的情况。

一个提议需要以下参数：

recipient 这个地址是，如果提议被接受，转移以wei为单位的amount的目的地址。

amount 如果提议被接受，需要转移多少wei的金额到recipient。

description 该提议的纯文本描述。

votingDeadline 一个unix时间戳，标记投票的结束时间。

open 布尔值，如果投票已经被记入是false，否者是true。

proposalPassed 布尔值，是否满足法定人数并多数人同意提议。

proposalHash 一个用来验证提议的哈希值。等于`sha3(_recipient, _amount, _transactionData)`。

proposalDeposit 在提交一个提议时，创建者必须要发送的最低押金（单位wei）。它来自在调用`newProposal`的`msg.value`；它的目的是防止垃圾提议。默认设置为20个以太币，但是提议的创建者可以发送更多押金。对于Slock.it，在GUI页面，提议会按照押金数量来排列显示，所以如果一个提议被认为很重要的话，提议的创建者可以支付更多的押金来宣传它的提议。如果满足法定人数，这笔押金会全部返回给提议的创建者，如果没有满足押金，这笔钱会保留在DAO中。

newServiceProvider 布尔值，如果提议用来指定新的服务提供方则为true。

splitData 这些数据用来分隔DAO。如果他们需要一个新的服务提方，这些数据从提议中收集。

yea 赞成提议的代币数。

nay 反对提议的代币数。

votedYes 一个简单的mapping用来检查一个代币持有者是否已经投赞成票。

votedNo 一个简单的mapping用来检查一个代币持有者是否已经投反对票。

creator 创建提议的代币持有者的地址。

分隔数据的结构体用于分隔DAO。它包括：

splitBalance 当前DAO的余额减去在分隔时提议的押金。

totalSupply 在分隔时DAO中存在的总的金额。

rewardToken 在分隔时原有DAO拥有的奖励代币金额。

newDAO 新的DAO地址（如果没有新建则为0）

这些是所有的成员变量，它们会在区块链上保存在智能合约中。这个信息可以在任何时间使用以太坊的客户端从区块链上读取出来。

现在我们来详细地讨论DAO合约中的函数。合约中的很多成员变量被地定义在其他三个合约中的某一个。

这里有一个特殊的函数，我们称之为结构体。它有作为DAO合约相同的名称。这个函数在DAO创建时，只执行一次。在DAO结构体中，以下变量将会被设置：

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

为了和智能合约交互，需要使用以下函数：

fallback function

fallback函数是一个没有指定函数名的函数。当合约接受到一个没有数据的交易时（单纯的价值转移），函数会被调用。这个函数没有直接的参数。fallback函数在预售阶段，会调用`buyTokenProxy`，并以发送者的地址作为参数。这会触发立即购买代币。为了保护用户，在预售阶段结束后的40天，这个函数会将接受到的以太币返还给发送者。这个函数被重新调用，使用`receiveEther`函数用来接受以太币作为DAO的押金。

payDAO

这个函数用来接受和跟踪DAO众筹产品产生的奖励，如果以太币已经被加入`rewards`则返回true。对于Slock.it来讲，这些奖励是在Slocks部署和使用的时候生成的。

receiveEther

一个简单的函数用来接受以太币。它什么也不做，只是当DAO接受到以太时返回true。

newProposal

这个函数用来创建一个新的提议。函数的参数如下：

recipient 在提议中以太币接受者的地址（必须是DAO地址自己，当前的服务提供方或者在`allowedRecipients`白名单里的地址）。

amount 发送给提议交易地址的金额，单位为wei。

description 提议的描述。

transactionData 提议交易的数据。

debatingPeriod 讨论提议的数据，一般提议至少要两个星期，如果是新的服务提供方至少是10天。

newServiceProvider 布尔值，定义提议是否是关于新的服务提供。

checkProposalCode

这个函数用来检查某个提议ID是否符合某一个交易。这个函数的参数如下：

proposalID 提议ID。

recipient 提议交易的接受者地址。

amount 发送给提议交易的金额，单位wei。

transactionData 提议协议的数据。

如果`recipient`，`amount`，`transactionData`匹配提议ID，这个函数会返回`true`，否者返回`false`。它用来验证提议ID是否匹配DAO代币持有者支持的对象。

vote

这个函数用于提议的投票。它的参数有：

proposalID 提议ID。

supportsProposal 布尔值用来表示代币持有者是否支持提议。

这个函数用来检查发送着是否已经投票和提议是否仍旧可以投票。如果两个条件都满足，它会记录投票到合约的storage中。被用于投票的代表会被锁定，意味着他们不能被转移，一直到提议被关闭。它会避免使用不同的发送地址投票数次。

executeProposal

这个函数可以被任何人调用。它记录投票数，用来检查是否满足法定数，如果通过会执行，除非它一个新的服务提供商的提议，否则它什么也不做。这个函数的参数有：

proposalID 提议ID。

transactionData 提议的数据内容。

这个函数检查投票的截止时间是否到达，和`transactionData`是匹配提议ID的。它会检查法定数(see Eq.1)是否被满足，是否提议被多数投票同意。如果是，执行提议，并返回提议的押金。如果法定法定数满足，但是提议提议被多数投票人否决，提议押金会被返回，提议关闭。

splitDAO

在一个新的服务提供商被提议时，在讨论阶段，代币持有者可以投赞成或者反对提议通过，这个函数被每一想要离开当前的DAO而去新的DAO（被提议的新的服务提供商）的代币持有者调用。这个函数新建一个新的DAO，并将转移部分以太币和部分奖励代币到新的DAO。参数有：

proposalID 提议ID。

newServiceProvider 新的DAO的提议提供商的地址。

经过合理性检查后（见代码），这个函数会新建一个新的DAO，如果它还没有使用`daoCreator`合约新建的话，更新分隔数据存储到提议中，并且存储新的DAO地址到分隔数据中。这个函数将会移动函数调用者的部分以太币，从原DAO到新的DAO地址中。以太币的数量记为Ξ_{sender}，用wei表示，计算如下：

$$\Xi_{\text{sender}} = \Xi_{\text{DAO}} \cdot T_{\text{sender}} / T_{\text{total}} \qquad (4)$$

T_{sender}是函数调用者的代币数量，Ξ_{DAO}是DAO在分隔时的余额。这实际上是被用于购买新建的DAO的代币，新的DAO的基金刚好就是原有的DAO的资金。另外奖励的代币R_{sender}也会被转移到新的DAO中去。计算方法如下：

$$R_{\text{sender}} = R_{\text{DAO}} \cdot T_{\text{sender}} / T_{\text{total}} \qquad (5)$$

R_{DAO}是原有的DAO在分离时的奖励代币总额。这些代币允许新的DAO获取这部分奖励，使用原有DAO的`getMyReward`函数。在这个过程的最后，这个发送者账户的所有的原有DAO代币会销毁。

transfer and transferFrom

这些函数重载了原来在`Token`合约里的实现。他们也调用了在`Token`合约里的`transfer / transferFrom`函数，但是他们额外地增加了转移关于这部分代币相关的已经支付的奖励，使用`transferPaidOut`函数实现。

transferPaidOut

当转移DAO代币使用`transfer`或`transferFrom`函数时，它更新`paidOut`数组用来记录已经支付的奖励金额P，计算如下：

$$P = P_{\text{from}} \cdot T_{\text{amount}} / T_{\text{from}} \qquad (6)$$

P_{from}是已经退还到发送`from`地址（发送者）的总的以太金额，T_{amount}是转移的总的代币金额，T_{from}是发送者`from`拥有的代币。

transferWithoutReward and transferFromWithoutReward

和`transfer`，`transferFrom`一样，但是会在之前调用`getMyReward`。

getMyReward

调用`withdrawRewardFor`，并以发送者为参数。它用来从`rewardAccount`中取回属于发送者的奖励部分。

withdrawRewardFor

这个函数用来取回在rewardAccount中的属于参数中的地址的部分。他们的奖励代币部分S，计算如下：

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

R_{DAO}是DAO的奖励金额，T_{sender}是发送者拥有的DAO代币。R_{sender}会一直是0，除非发送者是一个从原DAO分隔出的DAO，并且拥有它自己的奖励代币（从分离中）。Ξ_{reward}的以太数量是发送给DAO的代币持有者的，计算函数：

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

$\Xi_{\text{rewardAccount}}$是`rewardAccount`接受到的总的奖励，$R_{\text{total}}$是已经产生的奖励代币的总量，无论是分隔（`totalRewardToken`）还是$\Xi_{\text{paidOut[sender]}}$都是以wei为单位的金额，都已经支付到DAO代币持有者。奖励的代币会在8小节进一步阐述。

addAllowedAddress

这个函数增加一个地址到白名单`allowedRecipients`。它只能被服务提供商执行。

halveMinQuorum

最小最小的法定数在超过52周没有满足时，被minQuorumDivisor加倍。

numberOfProposals

返回已经新建的提议总数。

isBlocked

当作为参数的地址，在转移代币时，因为正在参与进行中的投票被阻止，返回true，否则返回false。

changeProposalDeposit

这个函数改变参数proposalDeposit。它只能被DAO，通过被多数代币持有者投票通过的提议去修改。

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);
}
```

这个合约用来管理奖励和extraBalance（如5节解释）。它有两个成员变量：

地址变量owner，是拥有提取账户金额权限的唯一地址（在我们的案例中是DAO），也可以使用payOut函数发送以太币到其他账户。

整型accumulatedInput，表示当前发送到这个合约的总的以太币数量（以wei为单位）。

当合约收到一笔无数据的交易（单纯的价值转移）时，回调函数被调用。这个函数没有直接的参数。当它被调用，它会计算收到的以太数量，然后将它存储到accumulatedInput。

函数payOut只能被owner（在我们的场景下指DAO）执行。它有两个参数：接受者和数量。它用来发送：recipient和amount。它用来发送以wei为单位的金额amount到接受者recipient，在DAO合约中被getMyReward调用。

7 Reward Tokens

在这节我们会描述奖励代币如何在合约中工作的。多数的内容已经被解释过，但是这里为了清晰起见会重述。

奖励代币用来区分在各种拥有奖励代币的DAO中发送给rewardAccount的金额。奖励代币只是在DAO分割时被转移的部分，他们从来不被任何一方拥有，不论是原DAO还是已经产生奖励代币的原DAO的分隔出的DAO。

奖励代币会产生，当DAO发生任何消耗以太币的交易时（除发送以太币到rewardAccount）。当DAO产品发送以太币返回到DAO时（比如当一个Slock发送 1% 的交易费到DAO中），这笔费用会和DAO拥有的其他的比特币保存在一起，但是rewards只是计算所有作为奖励收到的以太币。DAO会使用这些奖励去发起新的提议或者均等的分发给代币持有者（使用提议并被DAO代币持有者投票）。然后DAO的代币

持有者就可以要求他们为贡献原有的DAO（处理奖励代币）所获的以太币。为此，DAO会发送积累的奖励到`rewardAccount`，它会保存在`ManagedAccount`合约中。然后只有DAO代币的持有者才可以通过调用`getMyReward`函数来获取他们的代币。这些支付被map类型的`paidOut`记录，它用来记录DAO保存奖励代币和代币持有者是否获得他们的奖励份额，以及有哪些DAO的代币持有者已经获得了他们应得的奖励份额。这个过程保证了代币持有者，花费他们的众筹金额在建立产品的时候会获得奖励，确保他们即使在DAO分隔出去后依然能从这些产品中获得奖励。

8 Split

在这一节，我们将正式地描述在分割过程中的一些参数和它们的行为。

DAO代币总量`totalSupply`被定义如下：

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

T_i是地址i（余额`balances[i]`）所拥有的DAO代币数量。注意2^{256}是以太坊系统中可能存在的地址的总量。类似地，奖励代币的数量R_{total}被定义如下：

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

对于每一个获得获得通过的将以太币从DAO发送出去的提议，等同于被支出数量的奖励币被创建出来。

假设在分割期间，一部分DAO代币，假设数量为X，改变了服务提供商，并离开了DAO。新创建的DAO收到$X \cdot \Xi_{\text{DAO pre}}$，原DAO剩余以太币的一部分。

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

$\Xi_{\text{DAO pre}}$是原DAO在分隔前的以太币总额，$\Xi_{\text{DAO post}}$是原DAO在分隔后的以太币总额。

一部分的奖励代币以相似的方式被转移到新的DAO中。

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

R_{DAO}是DAO拥有的奖励代币的数量（在第一次分隔前，100%的所有奖励代币都属于DAO）。

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

新的DAO拥有的奖励代币的数量由R_{newDAO}表示。在分隔过程中，奖励代币的总量R_{total}保持不变，没有任何奖励代币被销毁。

原DAO中用于确认新服务提供商的代币会被销毁。因此：

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

这个过程允许DAO代币持有者在任何时间，不丢失任何未来奖励的情况下，获取他们的以太币。他们有资格，在即使他们选择离开DAO时，也可以获取以太币。

9 Updates

虽然被指定到某一特定以太坊区块链地址的合约代码不能被改变，但是可能仍然需要一位成员或者整个DAO以改变合约。正如上面介绍的，每位成员都可以分割DAO，并将自己的资金转移到一个新的DAO。他们可以将资金从自己分割出的DAO转移到另一个具有全新智能合约的新DAO。但是，为了整个DAO更新代码，某位成员可以创建一个新的具备所有必须特性的DAO合约，并部署到区块链上，提议把原有DAO中的所有以太币转移到这个合约中。如果提议被接受，整个DAO转移到这个新合约。为了在新的合约中使用相同的DAO代币，成员可以使用批准函数，给与新DAO转移代币的权利。在这个新的合约中，这一权利只有在受限的函数中可用，只有代币的所有者才能赎回代币。这一过程允许DAO在以太坊区块链上维持静态不可更改的代码，同时如果有需要，仍然可以升级DAO。

10 Acknowledgements

我要感谢Stephan Tual和Simon Jentzsch富有成效的讨论和修正，也要感谢Gavin Wood和Christian Reitwiessner帮助检查合约和开发我们写合约所用到的Solidity编程语言。

特别感谢Yoichi Hirai和Lefteris Karapetsas检查、改进文中的智能合约。

我也要感谢Griff Green检查和编辑这篇白皮书。

最后，我要感谢我们的社区给予的反馈、修正和鼓励。

参考文献

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.

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;
    }
}
```