AssetShare: A Secure Platform which Allows Multiple Parties to Own and Manage Shared Assets on a Blockchain

Alexandru Babeanu (4881133)
Jeroen Kloppenburg (4477960)
Maartje Veraart (4393791)
Panagiotis Reppas (5141435)
Pieter Kools (4134451)

April 15, 2020

1 Introduction

The overproduction of goods is most often accompanied by an inefficient usage time of products. That means that in many cases products are used for only a small portion of their life cycle. In order to solve this problem, a product could be shared among those who are willing to share the cost of the product.

We believe that in the near future there is going to be an increasing demand for sharing assets between multiple parties. It is obvious that this is becoming more of a trend with the advent of ride hailing and ride sharing applications. However, the same concept of sharing can be extrapolated to various other assets and can be applied in other contexts as well. Therefore, the need to manage such shared assets becomes more and more important.

AssetShare provides a secure platform which allows multiple parties to come together in order to own and manage shared assets. It ensures asset ownership and revenue distribution without the need for intermediaries, i.e. the platform provides necessary and sufficient proof of ownership for each share of the asset, as well as proof of entitlement for each share of the revenue. It also allows owners to make governance decisions involving the asset through a secure and transparent voting system.

The application's source code can be found at [1].

2 Project Description

In this section, we explain the problem that we intend to solve in more detail and describe the use case of the proposed application with a workflow.

2.1 Problem Statement

As previously stated, it is very common for people or businesses to pay for products and use them just for a small fraction of their lifetime. This affects businesses as well as individual consumers that own assets and cannot fully exploit them. In terms of economic efficiency it would be better for users if during the time these assets are idle, they would be shared with someone else who is willing share the cost for the product. For example, ten businesses with solar installations share a battery to store overproduction and to provide services on the energy markets. Or, five neighbors buy one washing machine that is also rented to others.

Currently, if people or businesses decide to collaborate and own an asset in common, the involvement of several other intermediaries is necessary. For example, a notary needs to draw up or certify contracts and banks are required for the financial transactions to be completed. All these entities demand fees for their services, hence further reducing the asset's revenue.

There is also the issue of trust. The asset owners have to rely on the competence and integrity of these fallible intermediaries.

We tackled these issues by creating a system that ensures asset ownership and revenue distribution without the need for intermediaries. Additionally, the users do not have to put their trust on anyone.

2.2 Stakeholders

Based on the description of our use case, the participants of our system can be categorized into the following types:

- **Owners:** The individuals, or the organizations that own shares of an asset.
- **Users:** The individuals or organizations that interact with the blockchain network. The users can transfer payment to an asset, or purchase shares of an asset and become an owner.

2.3 Workflow

Below is a workflow of the shared assets system explained. This is a short list of the actions a user may undertake when using our application:

1. A user publishes the asset they wish to share with other users.
2. The user selects the asset in which they are interested in. This does not necessarily have to be the asset they hold shares in.
3. Information of the asset becomes available, such as the number of shares each owner has, the treasury balance, or the revenue generated by the asset.

4. If a user owns shares in an asset, they are entitled to a portion of the asset income, which they can withdraw at the push of a button.

5. The user can buy or sell shares of the assets, and offer or demand a certain price for the amount of shares they offer or request.

6. A user can also publish proposals to change a feature of the asset manager, e.g. the amount distributed to the asset treasury, or to use treasury funds. This feature is only available on assets in which the user owns shares.

7. A user can vote on existing proposals, and implement them if they reach the approval threshold.

3 Requirement Analysis

In this section we briefly explain the requirements of our project. The detailed MoSCoW requirements analysis can be found in Appendix A. The following requirements have been determined:

- The application allows a user to publish an asset, or group of assets, on the blockchain, and initially have full ownership of it / them.
- Owners are able to offer shares of these assets for sale and anyone on the blockchain should be able to purchase them.
- The assets are able to receive payments and distribute this income periodically to the owners, according to their shares.
- The asset retains a portion of its income into a treasury, to be used for maintenance, operation costs, or expansion.
- The owners are allowed to publish proposals regarding the management of an asset, including the flow of funds in and out of the treasury. Owners can also vote on existing proposals.
- A user interface facilitates the interaction between users and assets.
- The interface can display financial statistics like the price of a share over time, the income generated by a share over time, or the user's return on investment.

4 Architecture

This section offers a detailed description of our architecture, design choices, and the technologies we use. Our application was developed using Aragon [2], a distributed application development framework for the Ethereum [3] blockchain.

4.1 Blockchain

What differentiates our app from other existing implementations is the use of blockchain technology [4]. There has been a lot of hype around this new technology over the past few years and in most of the cases it has not been justified. However, in our case blockchain technology has many properties that offer various advantages to our application.

Due to its distributed nature, blockchain removes the need for middlemen or trusted authorities to verify and guarantee ownership and transactions. As a result of this, our application is more efficient and less costly than other implementations. Additionally, consensus is reached among peers with equal authority, which guarantees that users will not exploit each other. Its transparency and immutability offer unprecedented levels of traceability and security. Since the validity of each operation can be mathematically proven, and because once an operation is performed the records cannot be altered, the system is protected against fraud and allows users to co-ordinate with each other securely, without having to trust one another or any additional third-parties.

4.2 Platform choice (Ethereum vs Hyperledger Fabric)

To start the implementation of the project, we first had to choose a platform that had the best fit with the requirements of our project. After doing our own research as well as consulting with our client, we narrowed it down to two options: Ethereum and Hyperledger Fabric. Both of these platforms are hugely famous for their unique structures. More so, both offer a lot of different features that can define our business model.

- Ethereum - Ethereum's first release date was in 2015 and as his creator described it in his whitepaper, it is an alternative protocol for building decentralized applications, with an emphasis on security, scaling and development time. As a permissionless blockchain, it has no restrictions regarding the identities of processors, thus everyone can start mining to create blocks. Ethereum's users can perform a range of different types of transactions using smart contracts. A smart contract is a computer protocol intended to digitally enforce a negotiated contract, they allow the performance of credible transactions without the need for third parties. Ethereum uses smart contracts that are written in Solidity programming language. Every action performed on the platform requires some quantity of computational power and time and miners must be paid for fulfilling the computational workload. For that reason a native currency known as Ether was developed.

- Hyperledger Fabric - Fabric was developed in 2016 [5]. Like other blockchain technologies, it has a ledger, uses smart contracts, and is a system by which participants manage their transactions. The characteristic that sets Hyperledger Fabric apart from other blockchain systems is that it is private and permissioned. In order for someone to be part of the network, he

must enroll through a trusted Membership Service Provider (MSP). Hyperledger Fabric's implementation of a smart contract is called chaincode and can be written in Go or JavaScript.

The decision not to use Fabric was based on a number of criteria. To begin with, we must take into account the fact that Hyperledger Fabric 1.0 was released in 2016. This means that, not many programmers use it and there is a lower number of users, uses, and documentation compared to Ethereum. Moreover, another important reason was the fact that Fabric was designed for permissioned blockchains, which we saw as a limitation in case we wanted to deploy our application on a public blockchain. Lastly, Ethereum has a native cryptocurrency which which our system can use as an abstraction for money.

4.3 Aragon

The Aragon framework offers various tools for developing Ethereum decentralised applications, including tools for setting up and deploying to devchains and testnets, a front-end development framework, integration with blockchain wallet managers, proxy contracts which facilitate upgrading contracts etc. Additionally, it offers various primitives useful for governance and access control, which might be useful for future work.

5 Implementation Details

Our implementation consists of two smart contracts written in Solidity and a front-end built with React JS, which integrates with our smart contracts through the Aragon framework. When deployed onto an Ethereum-based blockchain, the smart contacts will act as the back-end of our application, their main functionality consisting of storing and altering data while ensuring its integrity.

5.1 Back-end

5.1.1 Asset Factory

The *SharedAsset* contract implements all functionality related to a single asset, namely the share ownership and trading, income distribution, and asset governance. To allow users to deploy multiple assets on the blockchain, the *AssetShareApp* contract acts as a factory contract that deploys *SharedAsset* instances on request. Deploying our application to a blockchain is synonymous to deploying a single instance of the *AssetShareApp* contract. A call to the *createAsset* method on this instance will result in the deployment of a new *SharedAsset* instance onto the blockchain. The method takes a string as parameter, which, along with the address of the calling user, will be forwarded to the constructor of the *SharedAsset* contract. This constructor will set the asset description to the received string and the caller address as the sole owner of the asset (100% ownership).

The state of the *AssetShareApp* contract consists of an array of addresses. When a new *SharedAsset* instance is deployed, its address is added to the array. To retrieve these addresses, the contract offers two methods, *getAssetCount*, which returns the size of the array, and *getAssetByIdx*, which returns the address at the location in the array passed as argument, provided this location is valid.

5.1.2 Owning Shares

To keep track of an asset's owners, the state of a *SharedAsset* contract includes an array of addresses corresponding to the users who own shares in that asset. To keep track of how many shares each owner has, the contract state also includes a mapping from addresses to *Owner* data structures, which among other things contains the amount of shares owned by each user. A user is considered an owner if their corresponding amount of shares is higher than 0.

Shares are represented as unsigned integer values. The total amount of shares across all owners is equal to the constant value *TOTAL_SHARES*, which is currently set to 1.000.000. This constant also represents the maximum number of owners allowed by the contract. Its value can be increased for a higher granularity of shares and limit on the amount of owners, up to 2^{256}, but approaching this value is not recommended, since income distribution involves multiplying by the amount of shares owned by a user.

Besides the initial owner, a user can become an owner by purchasing shares from a preexisting owner. If an owner sells all their shares, they stop being an owner. All *Owner* structures keep track of the position of their corresponding address in the array of owner addresses, so that owner removal has constant complexity.

The following methods allow the retrieval of owner-specific data:

- *getOwnersCount* - returns the current amount of owners
- *getOwnerAddressByIndex* - returns the address of an owner by its location in the array of owner addresses
- *getSharesByAddress* - returns the amount of shares owned by the user with the given address
- *getSharesOnSaleByAddress* - returns the amount of shares offered for sale by the user with the given address

5.1.3 Trading Shares

To allow users to trade asset shares with each other, the *SharedAsset* contract provides the following methods:

- *offerToSell* - allows an owner to publish an offer for selling some of their shares
- *offerToBuy* - allows a user to publish an offer for buying shares from owners

- *sellShares* - given an active buy-offer, the caller can sell some of their shares in exchange for the offered amount of currency
- *buyShares* - given an active sell-offer, the caller can buy some of the shares offered in exchange for the asked price
- *combineOffers* - given an active sell-offer and an active buy-offer, if the amount offered by the buyer is higher than the price asked by the seller, it allows the publisher of one of the offers to combine the two by transferring as many shares and currency from one party to the other so that at least one of the offers becomes completed, i.e. the remaining amount of offered/requested shares reaches 0
- *cancelOffer* - given an active offer, deactivate it

An offer is represented by an instance of the *Offer* data structure, which contains the offer id, its type (buy-offer or sell-offer), its position in the array of active offers, the address of the seller and the address of the buyer, the amount of shares offered or requested, the amount of money offered or requested per share, and some bookkeeping values such as the creation date, completion date, and whether or not it was cancelled. The *SharedAsset* contract keeps track of offers through the use of two arrays, one containing all *Offer* structures ever created, and one containing the ids of the active offers, i.e. offers that have not been cancelled or completed. When an offer is created, it gets placed into the former array, and its id is set to its position in the array. Since the elements of this array are never removed nor do they change order, this id is constant. After that, the offer id is placed into the array of active offers. An offer is considered active if its id is in this array. When an offer is completed or cancelled, its id is removed from the array of active offers. This operation has constant complexity because the *Offer* structure keeps track of the location of its id in the array of active offers.

When publishing an offer, the user can specify an intended buyer or seller, for sell-offers and buy-offers respectively. When a user wants to trade shares through an offer that contains an intended buyer/seller, their address must match the one specified by the offer publisher.

For a user to be able to publish a sell-offer, or sell shares through a buy-offer, they must currently own the amount of shares they wish to sell. The *Owner* data structure keeps track of how many shares an owner has on sale, and the difference between this value and their total amount of shares represents the amount of shares they can (offer to) sell.

In order to publish a buy-offer, a user must transfer the total amount of currency they wish to exchange for shares. The currency will be stored in the contract balance and forwarded to users who sell shares through the buy-offer. If a user buys shares through a sell-offer, they have to transfer the full amount of ether corresponding to the amount of shares they're purchasing and the price asked by the seller.

5.1.4 Income Distribution

The *SharedAsset* contract offers a payable *payment* function which allows users to transfer currency to the contract. This operation simulates the revenue generated when the asset is used. This method will divide the income between the treasury and the owners, based on the *treasuryRatio* configuration variable. Distributing money to all the shareholders every time a payment is made to a contract would significantly increase the cost of sending payments to the contract. Since this cost would scale with the amount of owners the asset has, the contract may even become unusable due to out-of-gas exceptions. Because of these reasons, we have decided to make the income distribution pull-based instead of push-based. This is achieved by keeping track of the last known payout and the pending payout for every user. The method *withdrawPayout* allows users to withdraw currency from the contract balance, based on their shares and how much they have previously withdrawn. To make sure the income of a user is dependent on the amount of shares they had at the time the contract received a payment, and not on the amount they had when they withdrew their revenue, anytime a user's amount of shares changes, by buying or selling shares, the pending payout is updated to stay consistent with the payout the user is still owed.

5.1.5 Voting

To allow users to customize the behaviour of the shared asset or make use of the treasury funds the *SharedAsset* contract provides the following methods:

- *makeProposal* - allows an owner to propose a change to the contract configuration or a course of action
- *supportProposal* - allows an owner to cast a vote in favour of a proposed course of action
- *revokeProposalSupport* - allows an owner to withdraw their support for a given proposal
- *executeProposal* - allows a user to implement an active proposal if it has enough enough support
- *cancelProposal* - allows the publisher of an active proposal to deactivate it, and also allows any user to remove expired proposals

The application uses a *Proposal* data structure to represent proposals. This structure contains the proposal id, which is also its position in the list of all proposals, the address of the owner who published it, a string allowing users to explain the reasoning behind their proposal, a *Task* structure, which describes what happens if the proposal is implemented, a mapping from user addresses to the amount of support they offer to the proposal, the total support accumulated by the offer, a creation date, an expiration date, and a completion date. The

Task structure is used to describe the function that should run when the proposal is implemented, as well as the parameters that are passed to it. Currently, there are five actions which can be performed through voting:

1. Change the support threshold that has to be met by a proposal in order for it to be considered approved.

2. Change the asset description, which is represented by a string in the *SharedAsset* contract.

3. Change the ration of income distributed to the treasury.

4. Execute a function on a different contract, and if it is payable, use treasury funds to pay for it. This type of proposal allows owners to use the asset treasury to pay for services or maintenance.

5. Send an amount of currency from the asset treasury to a given address. This function allows the asset owners to divide either pay for goods or services using treasury funds, or split these funds amongst themselves.

Additionally, the application also offers the option of a free-form proposal, which does nothing when implemented, but allows owners to make any proposition and measure the support received from other owners.

The application uses a binary voting system, where an owner's vote on a proposal is "no" by default, unless they explicitly vote "yes". Each vote is weighted based on the shares the voter had at the time they voted "yes" on the proposal. This could cause problems if, for example, a malicious owner voted on a proposal, then transferred all their shares to a different account, then voted again. To avoid fraudulent behaviour such as this, owners are not allowed to support or publish proposals while having active sell-offers in the list of offers, and they are not allowed to sell shares while supporting proposals.

A proposal is considered approved if the accumulated shares of all owners who support it either reach or exceed the *proposalApprovalThreshold*. An approved proposal can stop being approved if enough of its supporters withdraw their support. As long as an approved proposal is active, any user can implement it by calling the *implementProposal* method. A proposal is considered active as long as it hasn't been cancelled, implemented or hasn't expired. A proposal becomes inactive after being implemented to prevent it from being implemented more than once.

5.2 User Interface

The user interface facilitates the interaction between users and the smart contract by displaying asset information in a user-friendly manner. The contract function calls are hidden behind UI elements like buttons and text fields. This section describes the front-end functionality and how it relates to the functionality of the smart contract.

5.2.1 Using the smart contract

Using the user interface the owner of an asset can manage their shares. The user can view the information of the asset, such as the amount of shares they own, or the revenue they are entitled to. Text inputs and button clicks are translated into method calls to the deployed contracts.

The interface makes use of a wallet manager to sign transactions. In order to call a method that modifies the contract state, a transaction has to be published on the blockchain network, signed with the private key of the user calling the method.

The methods of our smart contracts will emit events to notify external observers when the contract state has changed. The UI will listen for these events and update its display to correspond with the current state of the application.

5.2.2 Additional functionality

The front-end also offers some quality-of-life features, such as ordering offers by the cheapest price or highest offer, allowing users to interact with multiple offers simultaneously, or displaying various graphs and financial statistics related to assets and their owners.

When publishing an offer, if the *Autocomplete* check-box is ticked, the application will first complete as much of the new offer as possible, by using pre-existing compatible counter-offers, in a manner that generates the most income for the publisher. A buy-offer and a sell-offer are considered compatible if the price asked by the seller is lower or equal to the amount offered by the buyer, and if the intended counter-parties match the actual participants.

To calculate the return on investment of certain asset for a specific user, the interface retrieves the amount of currency that said user has spent on shares in a given asset, along with the amount they gained from selling shares and the passive income generated by their shares. The RoI is calculated by subtracting the amount spent from the amount gained, and dividing by the amount spent.

6 Evaluation

6.1 Functionality

The requirements seen in the requirement analysis and the 'must haves' and 'should haves' in the appendix are met. The application must allow a user to publish an asset on the blockchain. After adding an asset, owners will be able to offer shares of these assets for sale and anyone on the blockchain can buy them. Also the assets can receive payments and distribute the income amongst their owners. Thereby the asset can retain a portion of its income into a treasury. Proposals can be made by the owners regarding the management of an assets. All these requirements are facilitated by a user interface.

Although a lot of the requirements were implemented, improvements can always be made. For example a limit on the amount of offers in order to limit

spam is a requirement that is not met. This was a requirement in the 'could have' section of the MoSCoW analysis. In order to gain performance and lower security risks this could be an upgrade to the application.

6.2 Performance

Because we use no loops inside of our smart contract, all back-end functionality has constant complexity in terms of both time and gas cost. This also ensures that no out-of-gas exceptions can occur as more owners join a shared asset, or as more offers or proposals are published. Since the UI needs to display or interact with various lists, some its functionality does have linear time complexity, but these operations do not cost gas because the contract methods they employ are only executed locally.

6.3 Security

Our application uses the asymmetric encryption scheme built into the Ethereum protocol to perform user authentication. When a user wants to execute a method that modifies the state of a smart contract, they have to publish a transaction and sign it with their private key. When miners receive this transaction, they use the sender's public key to verify that they are who they say they are. This way, users cannot impersonate each other unless they know each other's private key. The methods of our smart contracts use this authenticated caller address to prevent fraud, i.e. a user cannot sell the shares of other users, or vote on another users' behalf, or cancel offers or active proposals that they did not publish themselves.

Our contracts cannot be driven into invalid states, either intentionally or accidentally. Each method performs all the required checks and actions to ensure that the state of the contract remains valid and no disallowed actions are performed. If a method is called with parameters that would cause the contract state to become invalid, an error is thrown, and the transaction is reverted, including all contract state modifications and currency transferred.

With regard to malicious behaviour, various rules are put in place to try to limit exploits. For example, in order to prevent the same shares to be used multiple times to vote on the same proposal, owners are not allowed to support or publish proposals while having active sell-offers in the list of offers, and they are not allowed to sell shares while supporting proposals. That said, there is one exploit that is not addressed by our implementation. If a user, or group of users, manage to pull enough shares to pass any proposal they want, they can deny the other owners from receiving income from the asset, by first setting the ratio of funds reserved for the treasury to 100%, and then siphoning the treasury funds through proposals for sending funds to a contract or account of their choice. To address this exploit, an additional layer of governance would have to be added to the system. Either through the use of a permissioned blockchain, or by vetting users based on some real-world information before allowing them to interact with the asset contracts.

Another security concern refers to the ability for users to spam the system with offers or proposals, in order to make it harder to use for other users. For the moment, the price of publishing any content onto the system serves as the only deterrent against such activities. An additional solution for this problem, which was envisioned but not implemented, would be to limit the amount of offers or proposals a user can have active at any given time. This way, users would be encouraged to remove older offers or proposals if they wish to publish new ones.

6.4 Unit tests

To test our application, we used the Truffle test suite as recommended by Aragon. The tests that we wrote can be found in the 'test' directory in the project. All tests were written in JavaScript and cover the main functionalities of the smart contract of the application. Since it is not possible to directly call internal contract functions from test code, we have extended the contract with a contract that defines public functions that redirect to these internal functions. This allows us to access internal functions from test code, without providing external access to these functions in the final application.

7 Future Work

1. Ethereum has set an upper limit regarding to the size of the contracts that can be deployed on the blockchain. The limit is 24 kilobytes and the idea behind this limitation is the fact that despite the fact that a call to a contract takes a constant amount of gas, the call can trigger O(n) cost in terms of reading the code from disk, preprocessing the code for VM execution, and also adding O(n) data to the Merkle proof for the block's proof-of-validity. While this is not an issue at current gas limits, it can cause inconvenience at higher gas limits [6]. Our system currently deploys two contracts that are under the limit. However, if further functionality is to be added to the platform it is necessary to break the contract into smaller ones.
2. To address exploit that allows majority share-holders to deny the minority shareholders their revenue, an additional layer of governance would have to be added to the system, so that minority shareholders can be protected from malicious users.
3. Introducing a limit on the amount of active offers or proposals a user can have would increase the system's protection against spam.

8 Conclusions

In this report, we have described the platform we have built, that asset owners can use to own and manage their assets. As presented in the problem

statement, the problem was the owners' desire to manage their shares without using a trusted third party. The platform that we built anticipated on this demand. Using Aragon, we built the application with several functionalities that support the demands of the owners. The first feature that we touched upon is that the users of the application can put out sell and buy offers, such that the owners of shares can sell and buy a parts of their shares. Furthermore, users can do proposals to change the management of the shares and only owners of the asset can participate in the voting. Also users could view the information of their shares. At last, the user could add more assets to the platform in order to manage all the assets they own.

References

[1] Assetshare code. https://github.com/babeanu-dorian/AssetShare.

[2] L. Cuende et al. Aragon. https://aragon.org, 2017.

[3] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[5] Linux Foundation. Hyperledger-fabric. https://hyperledger-fabric.readthedocs.io/en/release-2.0, 2015.

[6] V. Buterin. Contract code size limit. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-170.md, 2016.

A MoSCoW Requirements

A.1 Must have:

- The application will allow a user to publish an asset, or group of assets, on the blockchain, and initially have full ownership of it / them.
- The application will manage any number of owners, which can change over time, and each owner can posses a percentage of the asset, which may also change over time. An owner is defined as an account with more than 0 shares.
- There must exist some abstraction of money within the system, i.e. a currency. The system will keep track of how much money each user has and will allow monetary transactions between users.
- The owners will be able to offer portions of their shares for sale, and anyone on the blockchain, regardless of whether they own shares or not, should be able to purchase these offered shares. An account that is not an owner will become one upon purchasing any number of shares. An account will lose its owner status upon selling all its shares.
- The asset generates income, so the application will be able to receive payments.
- The application will distribute this income periodically to its owners, based on their respective shares.
- The application will manage a treasury, which siphons a portion of the income to use for management costs and other asset-related expenses. To make use of these funds, the owners must reach consensus through voting.
- The application will contain an interactive user interface, which will allow any user in the network to perform any of the aforementioned actions, provided they meet the required criteria, i.e. all actions are "owners-only", except for buying shares.
- The application front-end will also display various information and statistics about the state of the asset, namely how many shares each owner has, the amount of funds in the treasury, the market value of the shares over time, the ROI for individual users.

A.2 Should have:

- Both owners and non-owners should be able to make offers for buying shares. The application should handle the transfer of both money and shares.
- Owners should be able to trade with or transfer shares to specific accounts.

- Users should be able to cancel their own offers, and no shares or funds should be lost in the process. Users should not be able to cancel offers they did not publish themselves.
- Users should be able to partially complete offers, for example, if 10 % of the asset is for sale for 10 eth, a user should be able to purchase only 5 % for 5 eth.
- The owners should be able to make proposals for configuring the application (e.g. changing the ratio of funds diverted towards the treasury or the time between owner payouts) or for making use of the treasury funds (e.g. transfer some amount to a specific address).
- The owners should be able to vote on proposals, their votes being weighted by the number of shares they posses, and approve or reject proposed actions.
- Only an approved proposal can be executed, at most once, and the proposed action should be carried out by the application.

A.3 Could have:

- If a buying offer is made, and there are existing selling offers with a price lower or equal to the offered amount, the application could automatically complete (a part of) the buying offer.
- If a selling offer is made, and there are existing buying offers with an offer higher or equal to the requested amount, the application could automatically complete (a part of) the selling offer.
- The application could place a limit on the amount of offers and proposals a user can publish, in order to limit spam.
- The application could allow users to deposit funds directly into the asset treasury.
- The application could allow proposals for making use of the (payable) functionality of a different application on the blockchain.

A.4 Won't have:

- A moderation system that protects minority shareholders from abuse from majority shareholders.