Katallassos

A standard framework for finance

Bruno França, Sophie Radermacher and Reto Trinkler
Trinkler Software
company@trinkler.software

Version 1 – March 3, 2019

***Abstract*—Katallassos is a new blockchain that provides a standard way to build and deploy decentralized financial applications.**

It brings together all the components necessary for the backend of a financial application, namely: a high-performance consensus, an authenticated data feed system, a standard for financial contracts and connectivity to the rest of the blockchain ecosystem.

Katallassos enables and simplifies the creation of financial services that are non-custodial, trustless, fast, convenient and interoperable.

I. Introduction

Katallassos aims to create a standard framework for the next stage of decentralized finance. In order to do this, it combines a high-performance blockchain, built using an innovative consensus algorithm and virtual machine, with a runtime that allows easy creation of assets, a standard for the creation of algorithmic financial contracts, and connectivity to the outside world and other blockchains.

Katallassos appeals to both developers and users:

- For developers, Katallassos is a low-latency and high-throughput decentralized platform with native access to oracles, stable assets, connectivity to other blockchains and algorithmic contract templates that can be combined to create any type of financial contract. This allows developers to easily create decentralized financial services on top of Katallassos since custodianship, settlement and data feeds can be completely delegated to it.
- For users, Katallassos makes finance easy, using unprecedented speed, security, and risk management. Funds always remain under the user's control, empowering them to truly be their own banks. Settlement and data feeds are open and decentralized eliminating the need to trust exchanges. And, through a single Katallassos account, it is possible to interact seamlessly with all the financial services built on top of Katallassos (like futures and options exchanges, token exchanges, margin trading, short selling, collateralized loans, etc) and also with all the blockchains that connect to it.

In this paper we will introduce and describe every component of Katallassos and how they interact between themselves. The rest of the paper is divided as follows:

- In section II we will discuss the main building blocks of Katallassos.
- In section III we will give an overview of the runtime and operation of Katallassos.
- In section IV we will delve into the runtime and describe it in detail.
- In section V we will explore several examples of applications that can be built on top of Katallassos.

II. Building blocks

Katallassos is built on top of an infrastructure that is composed of six pieces:

- A blockchain development framework called Substrate,
- A consensus algorithm called Albatross,
- A virtual machine called Enso,
- A standard for the algorithmic description of financial contracts called ACTUS,
- An authenticated data feed system called Town Crier,
- And a heterogeneous multi-chain ecosystem called Polkadot.

We will now briefly explain each one of these components.

A. Substrate

Substrate is a software framework for blockchains created by Parity Technologies. It packs a series of tools, written in Rust, that enables developers to easily create a blockchain.

For developers who prefer simplicity over freedom, it is possible to generate a new blockchain with just a simple configuration file. While developers that prefer freedom can create their own consensus algorithms and runtimes from scratch.

Substrate is a combination of mostly two technologies, WebAssembly and Libp2p. The built-in WebAssembly interpreter enables developers to write their own ready-to-use modules in any language they wish, as long as it compiles down to WebAssembly, and Libp2p constitutes the bulk of the networking functionality.

For Katallassos we will use the most bare-bones version of Substrate, called Substrate Core, that provides only the networking, the WebAssembly interpreter and some other auxiliary tools and we will provide the consensus and runtime modules.

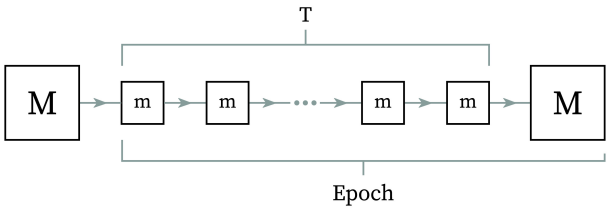
B. Albatross

Albatross [1] is a novel consensus algorithm, developed by Trinkler Software and Nimiq, that is inspired by speculative BFT consensus algorithms.

Speculative BFT is a class of classical consensus algorithms that have very high performance when compared to older algorithms like PBFT [2]. Speculative BFT consensuses have two modes of operation: an *optimistic* mode where nodes are assumed to be well-behaved thus resulting in greater performance, and the *pessimistic* mode where it is assumed that some nodes are malicious and the only goal is to make progress.

Albatross takes a *'trust but verify'* attitude to block production. Nodes are allowed to make updates to the state by themselves but other nodes can revert the update if it is not valid.

In Albatross there is a validator list that is selected at random from the set of all nodes that stake tokens. The validator list is changed every epoch, where an epoch is composed of T micro blocks and one macro block.



Validators take turns producing micro blocks, which are transaction-containing blocks that are signed by a single validator. Macro blocks do not contain any transactions [1], being instead used to change the validator set, and are produced with PBFT.

If all validators follow the protocol correctly, Albatross will produce blocks as fast as the network allows. Since macro blocks don't contain any transactions, there will be a small pause at the end of every epoch, but that downtime constitutes only a small percentage of the total time. So, in the optimistic case, Albatross will come close to the theoretical limit for a single-chain consensus algorithm.

However, if there are malicious validators they may misbehave. There are three ways in which they can do that:

- **Invalid blocks:** Validators may produce invalid blocks, in this case the other validators will simply ignore those blocks.
- **Forking:** Creating or continuing a fork will result in the stakes of the misbehaving validators being confiscated.
- **Delays:** If a validator, in his turn, does not produce a valid blocks after a predetermined amount of time, the other nodes will begin a *view change* protocol that will allow another validator to produce a block.

Albatross has some desirable features besides its low latency and high throughput. It allows nodes to bootstrap quickly by only requiring them to download the all macro blocks and the most recent micro blocks. It offers strong probabilistic finality, with a certainty of 99.9% being reached in only 6 blocks. And honest clients can get instant confirmation that their transaction will be accepted by directly asking for receipts from the validators.

C. Enso

Enso [3] is a general-purpose virtual machine for blockchains developed by Trinkler Software.

Blockchains can be seen as a distributed virtual machines, since they are designed to run a given application on a network of heterogeneous, and possibly malicious, nodes. We can represent a blockchain as a stack:

$$\begin{array}{c} \text{App data} \\ \hline \text{App logic} \end{array} \equiv \begin{array}{c} \text{State} \\ \hline \text{STF} \\ \hline \text{Consensus} \\ \hline \text{Networking} \end{array}$$

Where *STF* is the state transition function. Note that the application logic, also called the *runtime*, is embedded into the state transition function, while the application data is embedded in the state. For example, in the case of Bitcoin the application logic is that of a ledger and the application data is the set of all unspent transactions. This model works well for most cases but it is cumbersome to program and fails when it is necessary to update the application logic.

We take a different approach and have the state transition function be a general-purpose virtual machine. Then, the application logic and data can reside in the blockchain state:

$$\begin{array}{c} \text{App logic \& data} \\ \hline \text{Virtual machine} \end{array} \equiv \begin{array}{c} \text{State} \\ \hline \text{STF} \\ \hline \text{Consensus} \\ \hline \text{Networking} \end{array}$$

Enso is this virtual machine. Having a general-purpose virtual machine speeds up development, since only the state needs to be programmed, and allows for simpler and more fine-grained updates, because the state can be changed with simple extrinsics instead of forks.

Enso itself is a relatively simple virtual machine, inside of it everything is either an *object* or an *event*. Objects are entities composed of:

- **ID:** An unique identifier of the object. It can be any string.
- **Code:** A block of code containing functions that can be called by other objects.
- **Storage:** A data structure containing arbitrary information and that can be read and modified by the code.

While events are asynchronous function calls from one object to another and always have the following information:

- `priority`: The priority of the event, used for the event queue.

[1] We use same definition for transaction as the one taken by Parity. There are extrinsics, which are any input to the state transition function, and both transactions and inherents are mutually exclusive types of extrinsics. Transactions are extrinsics that are propagated through the network and signed. Inherents are neither propagated nor signed. An example of an inherent is a timestamp.

- `ID_to`: The ID of the object that will receive the event.
- `function_call`: The name of the function that will be called.
- `parameters`: A list of parameters that will be passed to the function.

The state of the blockchain is the set of all objects, and it is these objects that will contain all the application logic and data. *Everything is an object*.

The virtual machine itself is composed of just two components: the *event queue* and the *super object*.

The event queue is, as the name indicates, a queue for events. It is an ordinary priority queue and, when any event is created, it is added to this queue. Events with a higher priority go to the top of queue, while events with a lower priority go to the bottom.

The super object is a special object that is unique and can not be deleted or changed in any way. It is similar to the *super user* in Linux systems, in that it has complete control over the state. In fact, it is the only object that can change the ID, code and storage of other objects. The super object has the following interface:

- `Create(ID, code, storage)`: Creates a new object with the given ID, code and storage.
- `Delete(ID)`: Deletes object with the given ID.
- `Check(ID)`: Checks if any object exists with the given ID and returns the answer.
- `Request_object(ID)`: Returns the code and storage of the object with the given ID.
- `Change_ID(ID, new_ID)`: Changes the ID of the object with the given ID to new_ID.
- `Change_code(ID, new_code)`: Changes the code of the object with the given ID to new_code.
- `Change_storage(ID, new_storage)`: Changes the storage of the object with the given ID to new_storage.
- `Set_input(ID)`: Sets the input object to the object with given ID.

The *input object* is the object that is designated to receive extrinsics. An extrinsic in Enso is just a event like any other, but with the caveat that it must be sent to the input object. Any object can be the input object, the only requirement is that it designated as such by the super object.

A state transition in Enso looks like this:

1) Receive an ordered list of extrinsics.
2) Add the first extrinsic to the event queue.
3) The resulting event will be sent to the input object. The input object then may create more events, that in turn will also be processed and create more events, and so forth, until no more events are created.
4) Add the next extrinsic to the event queue and keep repeating steps 3 and 4 until there are no more extrinsics.

D. ACTUS

The *Algorithmic Contract Types Unified Standards* [4], or *ACTUS*, is a standard developed by the ACTUS Financial Research Foundation. It seeks to describe all possible financial contracts as algorithmic patterns of cash flows between two parties.

In ACTUS, any financial contract can be replicated as a combination of simpler contracts, called *contract types*. These contract types include basic financial contracts, like annuities and futures, and more exotic ones, like perpetual bonds and credit default swaps. In total there are 32 contract types, and together they form a taxonomy of financial contracts.

The contract types themselves are defined in algorithmic form. In other words, for each contract type there is a set of rules that, given some input parameters and external data, determine unambiguously the cash flows between the two parties to the contract. This allows them to be easily implemented in smart contracts. And, by combining different contract types, we can create any imaginable financial contract in algorithmic form.

There are five types of parameters that define a contract type: attributes, variables, contract events, payoff function and state transition function:

- Attributes are parameters that are static. They are defined when the contract is created and then they never change.
- Variables, as the name indicates, are parameters that are dynamic. At the time of the contract creation they are initialized to a given value, but they may change afterwards.
- Contract events are actions that cause a cash flow and/or a change in the variables. They may be scheduled, for example at the beginning of every month, or initiated by an external object, for example by one of the parties.
- The payoff function takes the attributes, the current variables and an event as inputs, and outputs a cash flow obligation from one party to another. The state transition function takes exactly the same inputs, but it outputs new values for the variables. Together, these two functions constitute the logic of the contract.

Contracts have two interfaces that enable it to interact with other objects: the risk factor observer and the child contract observer.

The risk factor observer allows the contract to request data from an oracle object, such data can be, for example, price information or interest rates.

The child contract observer allows one contract to observe the attributes, variables or events of another contract. This functionality is what makes it possible for several contract types to be joined together into more complex financial contracts.

Another useful characteristic of ACTUS contracts is that the cash flows between both parties of a contract, called the *creator* and the *counterparty*, can be *tokenized*. By this we mean that it is possible for users to have fractional ownership of a contract, thus it is perfectly possible to have the creator's or counterparty's cash flows (both positive and negative) be divided among several different users.

Tokenization also creates a simple way of transferring ownership of a contract. This feature is optional for any

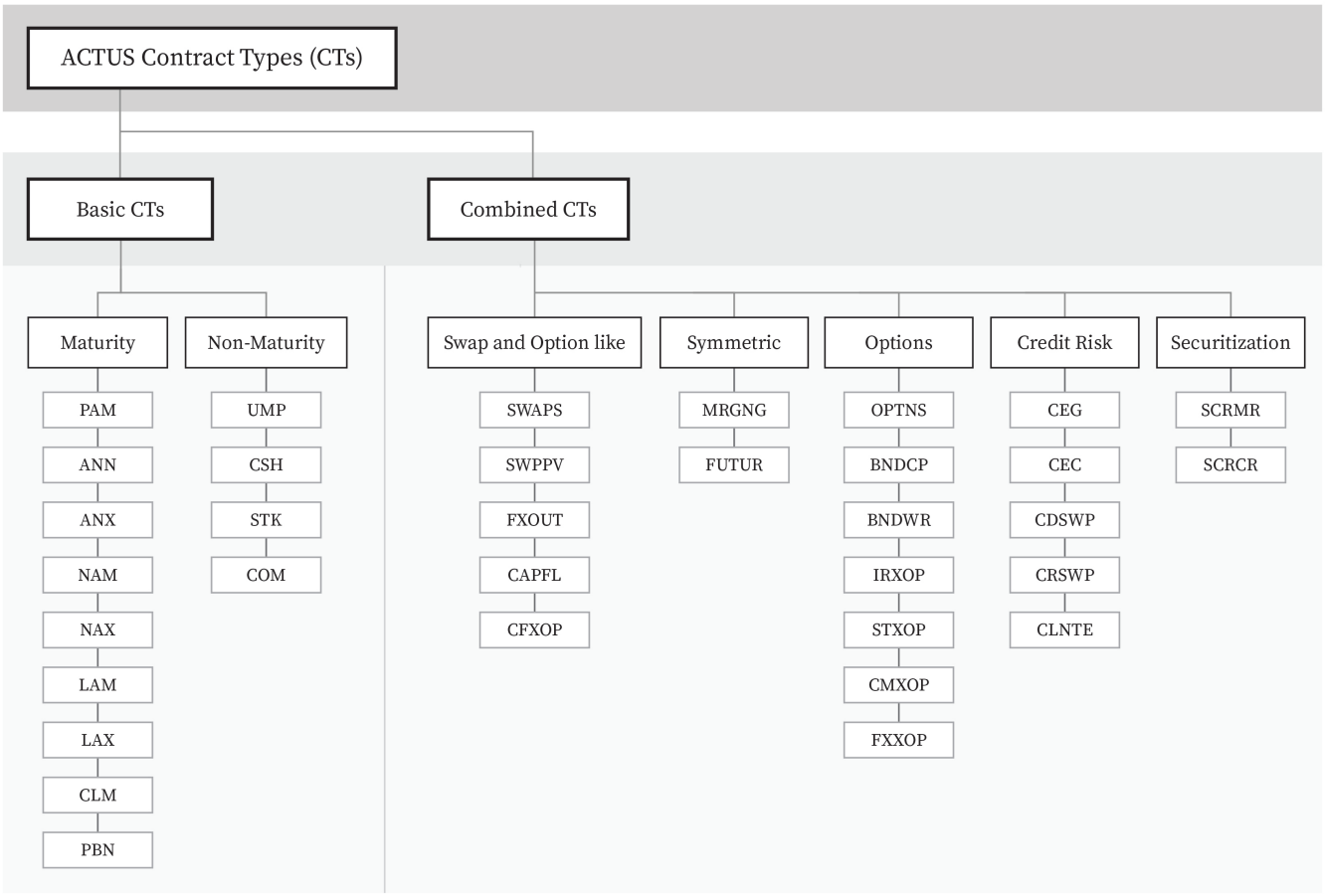


Fig. 1. The ACTUS taxonomy of financial contracts.

contract, since in some cases it may be undesirable, but when activated, it makes it possible for contracts to be sold in secondary markets.

E. Town Crier

Town Crier [5], developed by the Initiative for cryptocurrency and Contract (IC3), is an authenticated data feed system for smart contracts, also known as an *oracle*. It acts as a bridge between HTTPS-enabled websites and blockchains, and does so by taking advantage of a *trusted execution environment*, specifically the *Intel SGX*.

A trusted execution environment can be thought of as a space inside the CPU that allows programs that run inside it to be protected from other programs, the operating system and even from some types of hardware attacks. This space is called an *enclave*, and it is basically a *black box* inside which programs are certain to be executed correctly and with confidentiality.

An enclave can only use the CPU and the RAM by itself and needs to rely on the operating system for file and network access. However, by using public key cryptography, it can establish secure connections over the internet.

Another useful feature of enclaves is that they can provide a publicly verifiable proof that a given program was executed correctly and produced a given output. Such a proof is called an *attestation*.

In order to serve source-authenticated data to smart contracts, the Town Crier system only needs a specific smart contract, called the *oracle*, and a relaying server, called the *TC server*.

The oracle contract acts as the front-end for the blockchain, creating requests for data, verifying attestations from enclaves and distributing rewards to servers that provide data.

The TC server has two components: the relay and the enclave. The relay handles all the network traffic to and from the enclave, since the enclave itself has no networking capabilities. The enclave establishes HTTPS connections to websites and produces attestations.

The process works as follows:

1) The oracle contract produces a request for data. It does this by updating its state and signaling that it is ready to receive data from a TC server.
2) The relay, who periodically watches the blockchain, sees the data request and relays it to the enclave.

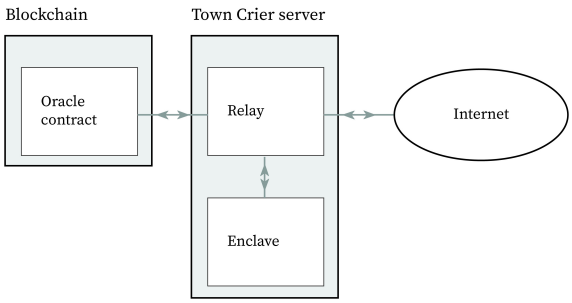


Fig. 2. The Town Crier system.

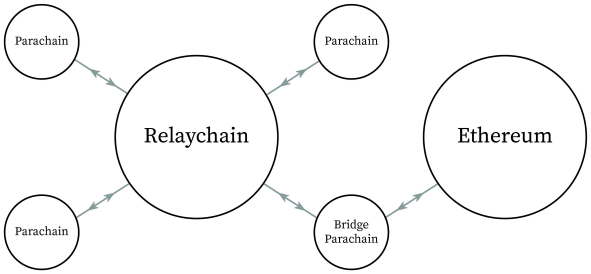


Fig. 3. Polkadot: relaychain and parachains

3) The enclave processes the request and initiates a HTTPS connection to the requested website.
4) The relay handles the traffic between the enclave and the website during the HTTPS session.
5) The enclave scrapes the website for the requested data and produces an attestation that the scraping was done correctly. Then, it sends the data and the attestation to the oracle.
6) The relay forwards the data and attestation to the oracle.
7) The oracle, after verifying that the attestation is correct, updates its state with the new data. Then, if appropriate, it distributes a reward to the TC server.
8) At any point, other contracts can fetch data from the oracle by requesting its more recent state.

Town Crier was recently acquired by Chainlink [6], a project that provides decentralized oracles for a variety of blockchains. Oracle contracts in Katallassos are updated by Chainlink nodes using the Town Crier protocol.

F. Polkadot

Polkadot [7] is a heterogeneous multichain framework introduced by Gavin Wood in 2016. It is a protocol that allows blockchains to exchange information but, unlike internet messaging protocols like TCP/IP, Polkadot also enforces the order and the validity of the messages between the blockchains.

Polkadot is composed of a central blockchain, called the *relaychain*, and a number of peripheral blockchains, called *parachains*, that connect to it. Parachains may outsource their consensus to the relaychain or have their own consensus algorithm and validators. The relaychain, as the name indicates, acts as a relay for messages between different parachains.

Connecting to Polkadot enables Katallassos to not only exchange information with other parachains but also for other parachains to transfer tokens to the Katallassos blockchain and vice-versa.

III. Overview

The Katallassos technology stack is illustrated in Figure 4.

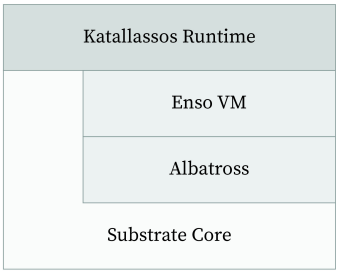


Fig. 4. Katallassos technology stack

Analyzing it we see that Substrate Core is at the bottom and is used for networking, module swaps and other auxiliary services. Albatross and Enso are, respectively, the consensus algorithm and the virtual machine, and they are implemented as modules in Substrate. Lastly, there is the Katallassos runtime on top.

It is the runtime that enables Katallassos capabilities and so, for the rest of this paper, we will focus mostly on it. Both Albatross and Enso are explained in detail in two other papers [1] [3].

The Katallassos runtime is composed of a set of objects and the interactions among them. Broadly speaking, the objects can be divided in two categories:

- **Kernel objects:** All the objects that are created at the genesis block and during updates to the blockchain. These are the objects that define the rules for how the runtime works and are unique objects, meaning that there is only instance of each object type. An example of a kernel object is the *governance object*, which handles updates to the blockchain.
- **User objects:** All the objects that are created by the users. Any user can create user objects from a predetermined list of object templates. For example, there is an *account template* and each instance of that template, created by the users, is an *account object*.

For a full description of the Katallassos runtime it is enough to outline all the kernel objects plus all the different object templates. Let us begin with the kernel objects:

- **Dock object:** It serves as the point of entry for extrinsics and is always the first object to be called, in other words it is the *input object*. It parses each extrinsic, verifies their signatures and then creates the events to the desired objects.
- **Authentication object:** It maintains a list of the IDs of all user objects and their corresponding authentication methods. The dock object calls this object to verify the signatures on transactions. Other objects may also call it when they need authentication services.
- **Schedule object:** It enables periodic, or scheduled, calls to other objects. Every block it receives from the dock object the current time and block number and then proceeds by calling any objects that are scheduled to be called at that particular time.
- **Instantiation object:** It creates all user objects. Users can call it to create a new user object from a list of object templates. It maintains the list of object templates and defines which parameters are allowed for the instantiation of those templates.
- **Native issuance object:** It is a special instance of the more general *issuance template*. It manages the native token (*KTS*) by maintaining a list of all object IDs and their corresponding balance. It also deals with transferring, minting and burning *KTS* tokens.
- **Governance object:** It implements whichever governance method is chosen to update the blockchain. Notably, it is the only object that possesses unrestricted access to the *super object*, thus allowing it to modify any part of the Katallassos runtime.
- **Consensus object:** It manages certain tasks related to the consensus algorithm. Namely, it maintains a list of validators and their staking deposits, applies slash inherents, distributes block rewards, etc.

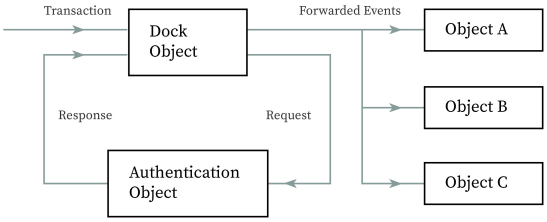
Besides these seven kernel objects, the Katallassos runtime is also composed of the following four user object templates:

- **Account object:** It is the most basic object in the runtime, being basically just an ID. Whenever an account is created, a corresponding entry is created in the authentication object with the authentication method and parameters chosen by the user.
- **Issuance object:** It creates and manages its own tokens. It can be user-controlled or automated, is capable of managing several different token types and supports both fungible and non-fungible tokens.
- **Oracle object:** It maintains a external data feed by serving as the interface for Town Crier servers. It creates requests for data, validates the data authenticity and distributes rewards to servers.
- **Contract object:** There are actually 32 different contract templates, one for each ACTUS contract type. All contract objects are capable of interacting with issuance objects, to create cash flows, and with oracle objects, to fetch data. It also maintains a list of all its owners, both creators and counterparties, for tokenization purposes.

The above list gives a rough description of all the objects but it does not shed much light on how they interact together. Even though it is not possible to detail all possible interactions in this paper, we will now discuss some of the most common ones.

A. Transaction docking

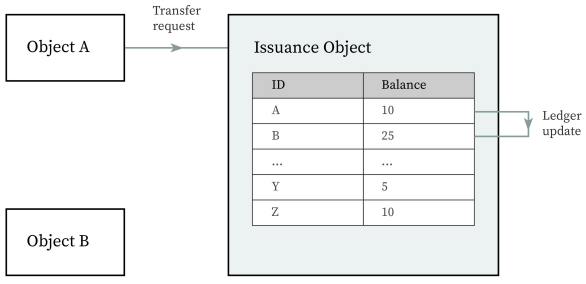
All transactions pass first through the dock object. Any transaction is just a *'bundle'* of function calls to other objects, and the dock object must first verify the validity of the transaction before creating the corresponding events for those function calls.



When the dock object receives a transaction it will first check that it is well-formed. Then it will call the authentication object to verify the signatures contained in the transaction. If, and only if, all signatures are valid then the dock object will create the requested events. Each of these events will include, as part of their parameters, the IDs of everyone that signed the transaction.

B. Token transfers

Like ERC-20 tokens in Ethereum [8], tokens in Katallassos are managed by a single object. In other words, instead of every account storing the balance of every token that it owns, for each token there is a single object that stores the ID and balance of every account that owns those tokens. These objects we call *issuance objects*.

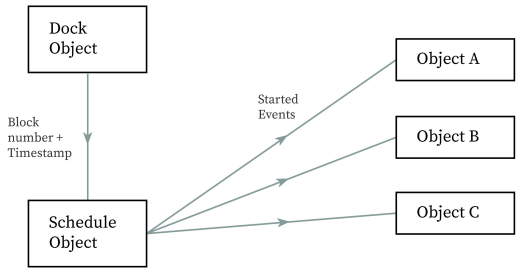


When an account wants to transfer some tokens to another account, it just sends a request to the issuance object of that

token. The issuance object will then update its internal ledger to reflect this change.

C. Schedule functions

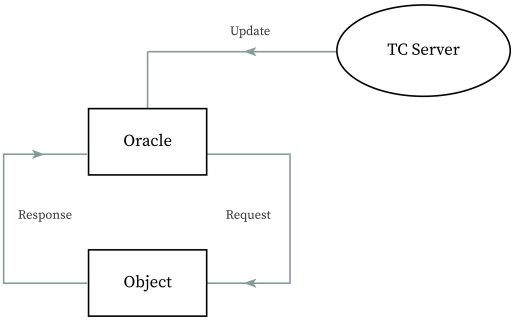
It is useful to have functions that are started at predefined times. For example, we may want an oracle that is updated every minute. In order for this to happen, some object needs to call the oracle object every minute, so that it can accept updates to its state. That object is the schedule object.



Every block the dock object sends the current block number and time to the schedule object. The schedule object has a list of functions, IDs and conditions. Upon receiving the block number and time, it goes through this list and, if any of the conditions is satisfied, it triggers the corresponding function at the corresponding ID.

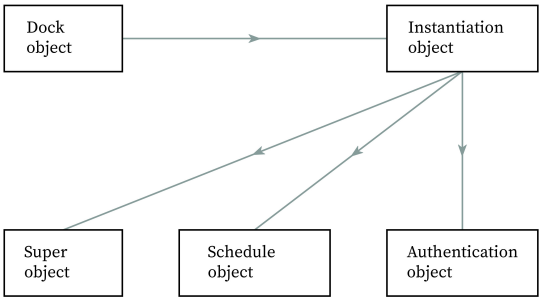
D. Oracles and data feeds

Each oracle contract only maintains one specific data feed. When they request an update, a Town Crier server can send a transaction containing the update and potentially receive a reward. Other objects can send a function call requesting data from the oracle and it will respond back with the most recent data in its state.



E. Template instantiation

Users can create new objects using the instantiation object. Through the dock object, an user can request the instantiation object to create a new object from a list of templates. The instantiation object then sends the necessary events to the super object and also, in some cases, to the authentication and schedule objects.



F. Governance

The main purpose of the governance object is to control who can have unrestricted access to the state, and especially to the super object. It will receive calls from the dock object and, given its internal logic, it will decide if the calls are to be forwarded or not.

IV. Specification

In this section we will detail the runtime of Katallassos, beginning by giving some general remarks and then by describing the kernel and user objects. Our focus will be on the objects interface and function.

A. General

1) Reserved namespace: There is a reserved *namespace* for the kernel objects. The namespace consists of all the IDs that begin with *'KTS'*. For example, the dock object will have the name *'KTS_Dock'*.

User objects are not allowed to have an ID that begins with *'KTS'*. This is enforced by the instantiation object, who will not accept any requests to create an user object with an ID in that namespace.

2) Transactions: A transaction is a bundle of function calls to objects. Users can join several different function calls into the same transaction and sign them. The function calls are only forwarded by the dock object if all the signatures are valid. A transaction has the following format:

- **[ID]:** A list of the IDs that are authorizing this transaction.
- **[Function calls]:** A list of the desired function calls.
- **Window:** Determines the time window in which this transaction is valid. For example, if the window is $[2000, 2050]$ then the transaction will only be accepted between the block number 2000 and the block number 2050.
- **Nonce:** A long integer chosen by the users. Together with the validity it prevents *replay attacks*.
- **[Signature]:** A list of the signatures for this transaction, one for each ID.

Where [.] represents a list. A transaction can also be thought of as a *wrapper* to a series of function calls. These function calls have the following format:

- `ID_to`: The ID of the destination object.

- `call_function`: The function to be called at the destination object.
- `[user_parameters]`: A list of user-provided parameters to be passed to the function.

3) Origin ID: All function calls, except the extrinsics fed into the dock object, will include in their parameters the field `ID_from` which is the ID of the object that originated the function call.

Adding this information to every function call allows objects to have functions that can only be called by certain objects. For example:

- Internal functions that can only be called by other functions in the same object,
- Kernel functions that can only be called by kernel objects,
- User-forbidden functions which are functions that do not accept calls from the dock object, thus they cannot be called by users,
- User-only functions which are functions that are meant to be called by users and as a result only accept calls from the dock object.

4) Extrinsics order: A block is composed of several extrinsics, either transactions or inherents, and they are fed into the dock object into a specific order:

- `Timestamp:={time, block number}`: An inherent that contains timing information. It is forwarded to the schedule object.
- `Slash:={ID}`: An inherent that is used to slash the stake of a misbehaving validator. It is forwarded to the consensus object.
- `Seed:={seed}`: An inherent that contains a random seed used to select a new validator set. It is forwarded to the consensus object only in macro blocks, otherwise it is ignored.
- `Transactions:={[transactions]}`: The list of transactions. They are only sent in micro blocks.
- `Reward:={ID}`: An inherent containing the ID to which the block reward is paid to. It is forwarded to the consensus object.

B. Kernel objects

1) Dock: The dock object acts as the point of entry for extrinsics and its interface only has one function:

- `input(extrinsic)`: Parses the extrinsic and verifies its validity. If it is a transaction then it calls the authorization object to verify the signatures and, if the all signatures are valid, then forwards the function calls to the correct objects.

For each function call in the extrinsic a new event is created by the dock object with the format `e(priority, ID_to, call_function, ID_from, [auth_ID], [user_parameters])`, where `[auth_ID]` is the list of every ID that signed the transaction.

2) Authentication: The authentication object verifies authentication proofs provided by the users. To do so it maintains the following internal key-value store, called the *authentication registry*:

Key	Value
ID	(Method, [Parameters])

Where method is any authentication method supported by the authentication object. It can be a signature scheme like ECDSA, Schnorr or BLS, or a multisignature scheme, or a hash-lock, or even a zero-knowledge proof system. The parameters are any information necessary to authenticate proofs. For example, in the case of ECDSA the parameters would be the public key.

The next functions form the interface of the authentication object:

- `verify([message, ID, proof])`: Goes through the list verifying that each proof is a valid authentication of the message by the corresponding ID. Accepts calls from any object.
- `method(message, proof, parameters)`: General function type for authentication. There is one instance for each different authentication method. It is an internal function, and as such it accepts only calls from the authentication object.
- `add_key(ID, method, parameters)`: Adds the ID with given method and parameters to the authentication registry. Does not accept calls from the dock object.
- `change_key([auth_ID], ID, method, parameters)`: Changes the authentication method of the object with given ID. Accepts calls from any object but, if the call originates from the dock object, it will only be accepted if ID $\in$ [auth_ID].
- `delete_key([auth_ID], ID)`: Deletes the entry in the authentication registry with given ID. Accepts calls from any object but, if the call originates from the dock object, it will only be accepted if ID $\in$ [auth_ID].

3) Schedule: The schedule object serves to make function calls to other objects at regular intervals. To achieve this, it maintains an internal key-value store called the *schedule registry*:

Key	Value
(ID, Function)	Condition

The key identifies which function needs to be called at which object, and the value states under which conditions the function call will be triggered.

Its interface is composed of the following functions:

- `init(block_number, timestamp)`: The main function of the schedule object, it is called once every micro block by an inherent. It goes through the entire schedule registry and checks if block_number and timestamp satisfies any of the conditions. If it does, it triggers the corresponding function call.
- `add_key(ID, function, condition)`: Adds the given ID, function and condition to the schedule registry. Does not accept calls from the dock object.

- `delete_key(ID, function)`: Deletes the entry in the schedule registry with given ID and function. Does not accept calls from the dock object.

4) Instantiation: The instantiation object creates new user objects from a set of object templates. Its interface consists of only two functions:

- `create([template_ID, parameters])`: For each item in the list it checks if the requested ID is available and, if they are all available, then instantiates the requested objects from template_ID with the given parameters. Accepts calls from any object.
- `template_ID(parameters)`: General function type, one exists for each object template. It instantiates a new object with the given parameters. It is an internal function.

5) Governance: The specific implementation of the governance object will depend on the governance method used. So, it is not possible for us to give a description of the object that will be valid in every case.

However, we will exemplify the simplest case, which is when one single ID has full control over the governance. This hypothetical *dictatorship* object would have the following interface:

- `transmit([auth_ID], ID_call, function_call, parameters)`: If dictator_ID $\in$ [auth_ID], it forwards the requested function call. Where dictator_ID is hard coded into the function. This function basically gives dictator_ID full access to the entire state.

6) Native issuance: The interface for the native issuance object is exactly the same as any other issuance object (see IV-C2), since it is just an instance of the issuance template.

However, we can be more specific regarding its specification:

- There is only one asset_ID, which is the native token *KTS*.
- The `mint` and `burn` functions only accept calls from the consensus object.

7) Consensus: The consensus object deals with all the tasks related to the validators, specifically managing stake deposits, collecting fees and distributing block rewards. To do so, the consensus object has a key-value store of all potential validators, called the *validator registry*:

Key	Value
ID	(deposit, validating key, status)

The key is just the ID of the account that deposited the stake. The value contains the amount deposited (in *KTS*), the public key used to validate blocks and the status. The status just indicates if the validator is currently active or not.

- `stake([auth_ID], ID, amount, validating_key)`: Transfers the given amount of *KTS* to the consensus object. If successful, it adds ID, validating_key and amount to the validator registry. It accepts calls from any object, but only if ID $\in$ [auth_ID].
- `restake(ID, new_validating_key, signature)`: Calls the authentication object to check if signature is valid. If it is, it replaces the validating key of ID.
- `unstake([auth_ID], ID)`: Returns deposit back to ID and removes the corresponding key from validator registry. It accepts calls from any object, but only if ID $\in$ [auth_ID].
- `fee([auth_ID], ID, amount)`: Transfers a given amount of *KTS* to the consensus object. If successful, updates block reward value. It accepts calls from any object, but only if ID $\in$ [auth_ID].
- `slash(ID)`: Divides the deposit amount of the given ID between all other active validators and deletes the corresponding key from the validator registry. Can only be called with an inherent.
- `reward(ID)`: Transfers the block reward to the given ID. Can only be called with an inherent.
- `change_validators(seed)`: Produces a new list of active validators from the seed and updates the registry accordingly. Can only be called with an inherent.

C. User objects

1) Account: Accounts are the object most utilized by users and also the simplest. They are basically just an ID. All the functionality normally associated with accounts, like transferring tokens and entering into contracts, is provided by other objects.

The interface of an account has a single function:

- `self_destruct()`: Deletes this object and all data associated with it.

2) Issuance: Issuance objects create and destroy tokens, and maintain a list of everyone who owns tokens. Each issuance object can have several different token types, each identified with asset_ID. Because of this property, issuance objects can support both fungible and non-fungible tokens.

Its interface is composed of the following functions:

- `mint(owner_ID, asset_ID, amount)`: Creates a given amount of tokens of type asset_ID and deposits them in owner_ID. Depending on the option chosen, it can be called by no one, by a predetermined set of IDs or it can be scheduled.
- `burn(owner_ID, asset_ID, amount)`: Destroys some amount of tokens of type asset_ID in owner_ID. Depending on the option chosen, it can be called by no one, by a predetermined set of IDs or it can be scheduled.
- `transfer(owner_ID, destination_ID, asset_ID, amount)`: Sends a given amount of tokens of type asset_ID from owner_ID to destination_ID. Accepts calls from any object but, if the call originates from the dock object, it will only be accepted if owner_ID $\in$ [auth_ID].
- `check(owner_ID, asset_ID)`: Returns the balance in asset_ID tokens of owner_ID. Does not accept calls from the dock object.

- `self_destruct()`: Deletes this object and all data associated with it. Depending on the option chosen, it can be called by no one, by a predetermined set of IDs or it can be scheduled.

3) Oracle: Oracle objects are the front-end to the Town Crier system. Their main purpose is to maintain a data feed. Such data feed can be any tuple, as long as it includes a timestamp:

```
Data:={a, b, ... , timestamp}
```

Oracle objects have the following interface:

- `request()`: Turns on a flag stating that it will allow updates to the data feed.
- `update(value, timestamp, proof, receiving_ID)`: Updates the oracle data feed with given value and timestamp, after verifying the accompanying proof. It can provide a reward to receiving_ID. It accepts calls from any object, but it may be permissioned.
- `fetch()`: Returns the latest value and timestamp.
- `set_reward(issuance_ID, asset_ID, amount)`: Sets the reward per update to amount of asset_ID at the object issuance_ID. Depending on the option chosen, it can be called by no one, by a predetermined set of IDs or it can be scheduled.
- `self_destruct()`: Deletes this object and all data associated with it. Depending on the option chosen, it can be called by no one, by a predetermined set of IDs or it can be scheduled.

4) Contract: Contracts are any financial contracts that are a part of the ACTUS standard. There are 32 different contracts, so it is not possible for us to give detailed description of every single one but, the interfaces of all contracts follow the same pattern.

The interface of a contract has two main functions: 1) processing events and 2) managing ownership of the contract. This results in the following general interface:

- `event(parameters)`: General function type, it is any event of the associated ACTUS contract type. It can be triggered by a user or by the schedule object. When triggered, the contract will update its internal state, fetch data from an oracle, observe another contract and/or create a cash flow.
- `transfer(owner_ID, destination_ID, position, amount)`: Transfers fractional ownership of the contract, corresponding to either the creator or counterparty position. Accepts calls from any object but, if the call originates from the dock object, it requires authorization of both owner_ID and destination_ID. Depending on the contract parameters, ownership transfer may not be allowed for the creator, the counterparty or both.
- `check(owner_ID, position)`: Returns the ownership amount of a given position by owner_ID. Does not accept calls from the dock object.
- `self_destruct()`: Deletes this object and all data associated with it. It can be scheduled or triggered by a contract event.

Some explanation is needed about the ownership mechanism. In order to support tokenization contracts have some of the functionality of a issuance object. Specifically, all contracts have exactly two different asset IDs, one for the creator position and another for the counterparty position. Also, each of these asset IDs has exactly one token unit. So, if someone holds 0.2 counterparty tokens, that means that he owns 20% of the counterparty position.

Contrary to the tokens of issuance objects, which are just symbolic representations of external assets, contract tokens have a more active role inside Katallassos. Contract tokens give whoever holds them a share in the future cash flows (positive or negative) generated by the contract.

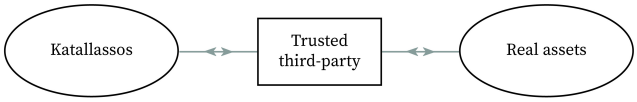
A cash flow is executed by a contract by directly calling an issuance object or another contract object. For example, imagine that Alice has 0.2 creator tokens, Bob has 0.8 creator tokens and Charlie has 1 counterparty token. If a contract generates a cash flow of 100 *KTS* from the creator to the counterparty, it will call the *KTS* issuance object to transfer 20 *KTS* from Alice to Charlie and 80 *KTS* from Bob to Charlie.

V. Functionality

Developers can build an endless variety of financial services in Katallassos by combining different ACTUS contracts. In this section we will give a few examples of the financial applications that can be created.

A. Asset-backed tokens

Issuance objects can be used to create tokens that represent any asset, all that is needed is a trusted entity to hold custody of the underlying assets and to allow the exchange between the token and the asset. This is one of the methods used to create stablecoins.



Any asset can be used to create asset-backed tokens, for example:

- Fiat currencies like US Dollar, Euro, Swiss Francs, etc,
- Stocks of exchange-traded companies,
- Commodities like gold, silver or oil,
- Land, houses and other real estate.

By far, the most useful asset-backed tokens are the ones backed with fiat, since they can be used as a settlement currency for financial contracts. But other assets also open interesting possibilities, like using stock-backed tokens to create a Katallassos stock exchange, or using real-estate-backed tokens to create a land registry.

Some critics point out that asset-backed tokens are centralized, but the matter of fact is that there is no better option.

For tangible assets, like commodities and real estate, no blockchain can hold custody of them, so a central entity is required. *Physical assets can not be stored in a blockchain.*

For intangible assets, like stocks and currency, it is technically possible to store them in a blockchain. But, by definition, these are centralized assets. A central bank controls the currency that it issues, and a company controls its own stock.

B. Transfers

Transfers are the simplest financial service and also one of the most basic functions for any blockchain. The convenience and usability of transfers in Katallassos matches, or even exceeds, that of banks, online payments systems and other cryptocurrencies:

- Variety of currencies: Katallassos supports transfers of both fiat currencies, through asset-backed tokens, and cryptocurrencies, through Polkadot connectivity.
- Human-memorable addresses: Accounts in Katallassos can have any unique string as their address, thus being more user friendly than other blockchains and banks.
- Variety of authentication: Katallassos supports a variety of authentication methods and lets its users pick the method that they prefer for their own account.
- Speed: Katallassos' consensus algorithm, Albatross, can finalize transactions in just a few seconds.

C. Token exchange

Many blockchain projects revolve around doing exchanges between different tokens. In Katallassos, a token exchange can be executed with a single contract, called a *foreign exchange contract*.

If Alice and Bob want to exchange two types of tokens, they only need to create the contract. Afterwards, it will transfer the respective tokens out of Alice's and Bob's accounts and transfer them to their new owner.

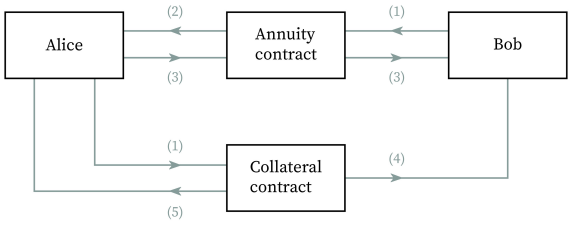
The foreign exchange contract is *atomic*, meaning that it either completes successfully or it does not happen at all. There is no risk for any of the parties involved.

Given that Katallassos supports both asset-backed tokens and cryptocurrency tokens (of any blockchain that connects to Polkadot), in addition to its native token *KTS*, the token exchange contract can be used in a variety of interesting situations:

- Fiat-KTS: Buying KTS inside Katallassos using a fiat-backed token.
- Fiat-Stock: Buying and selling stocks using a fiat-backed token, akin to a stock exchange.
- Fiat-Crypto: Buying and selling cryptocurrencies with fiat.
- Crypto-Crypto: Exchanging different cryptocurrencies.

D. Collateralized loans

More complex services can be constructed, for example collateralized loans. Imagine Alice has a house, which is represented in Katallassos by an asset-backed token, and she wishes to ask Bob for a loan while giving her house as collateral. To do this, Alice and Bob first create two contracts, an annuity contract and a collateral contract, that will codify the terms of the loan.



After creating the contracts, the following series of cash flows will happen:

1) The loan amount from Bob's account and the collateral from Alice's account are transferred into the corresponding contracts.
2) The loan amount is transferred to Alice's account.
3) Periodically, payments are transferred from Alice's account to Bob's account.
4) If at any time, Alice does not have enough money in her account for the loan payment, the collateral is transferred to Bob.
5) If every loan payment is made, at the end of the loan contract the collateral is returned to Alice's account.

Note that Alice and Bob do not need to interact with the blockchain after the contracts are created. All the cash flows are initiated and managed automatically by the contracts.

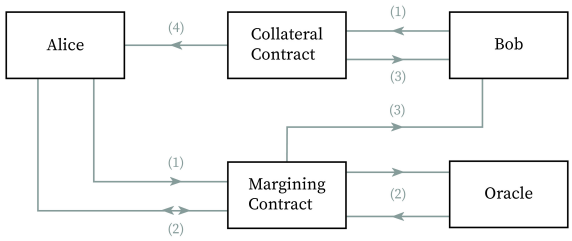
E. Margin trading

Margin trading is the act of borrowing money to buy assets and is widely used in all areas of finance.

In margin trading borrowers are required to maintain the net value of their position, meaning the difference between the value of the assets and the value of the loan, at a constant value. This value is called the *margin*.

This financial service can also be done in Katallassos, and is in fact similar to collateralized loans. Imagine Alice wants to

buy *KTS* on margin and to do so she will borrow money, in the form of *KTS* tokens, from Bob. Alice will maintain her margin using fiat-backed tokens. They will create two contracts: a collateral contract and a margin contract.



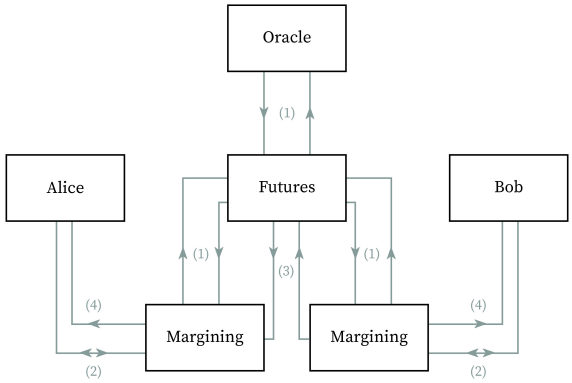
An oracle for the price of *KTS* is also necessary for adjusting the margin amount. With the contracts created, the following cash flows will take place:

1) The borrowed *KTS* is transferred from Bob's account into the collateral contract, and the margin amount is transferred from Alice's account into the margining contract.
2) Periodically, the margining contract will query the oracle for the *KTS* price and adjust the amount of margin accordingly, transferring cash in and out of Alice's account to maintain the necessary margin.
3) When Alice decides to terminate her position, or when she is no longer able to maintain the margin, the collateral and the margin are transferred into Bob's account.
4) If, when her position is terminated, Alice has made a profit then part of the collateral will be transferred into her account.

In this case we exemplified margin buying, but a similar scheme can be used for margin selling, also known as short-selling.

F. Futures

Futures are one of the most versatile financial contracts, allowing anyone to speculate not just on assets, but on practically anything, for example commodities, stocks, bonds, cryptocurrencies, fiat currencies, indexes, interest rates, energy, weather, etc. As long as there is some publicly available numerical value, a futures contract can be created for it.



Imagine Alice and Bob want to speculate on the price of oil. First, they need to see if an oracle for the price of oil exists in Katallassos, this is necessary for the contract. Then, Alice and Bob create three contracts: a futures contracts and two margining contracts, one for each of them.

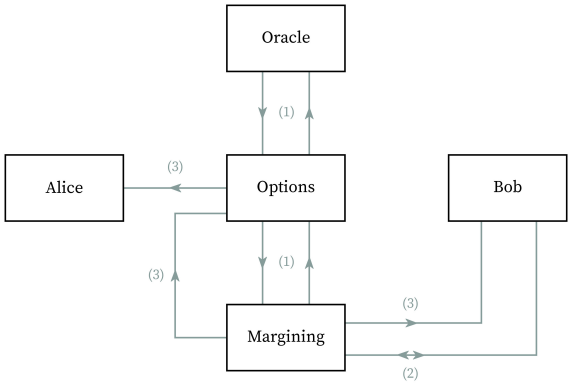
After the margin amount gets transferred into the margin contracts, the rest of the exchange proceeds as follows:

1) At regular intervals, the futures contract fetches the current price of oil from the oracle and updates its state to reflect the profit and loss of each of the parties. The margining contracts also observe the futures contract and adjust the amount of margin required.
2) The margining contracts transfer cash in and out of Alice's and Bob's account to maintain the necessary amount of margin.
3) When the futures contract ends, it will settle by making a transfer from one margining contract to the other.
4) Then the margin contracts will terminate and return their deposits to Alice's and Bob's accounts.

G. Options

An option is a contract that gives one party the option to buy or sell a given asset at a predetermined price. One party will pay a fee upfront in order to have that option, while the other party will receive that fee. So, the party that buys the option only has upside and the party that sells the option only has downside.

Like futures, options can also be used to speculate on almost any asset. The scheme for options in Katallassos is also very similar to the futures scheme.



Imagine Alice wants to buy an option on the price of gold from Bob. Alice will pay Bob for the option and, at the same time, an options contract and a margin contract will be created. The next cash flows will then happen:

1) Periodically, the options contract requests the current price of gold from the oracle and updates its state to reflect the profit and loss of each of the parties. The margining contract observes the options contract and adjusts the amount of margin required.
2) The margining contract transfers cash in and out of Bob's account to maintain the necessary amount of margin.

3) When Alice exercises the option, or when it expires, it settles by transferring the necessary amount from the margining contract to Alice's account. The rest of the deposit is returned to Bob's account.

VI. Conclusion

In this paper we introduced Katallassos, a new blockchain designed purposely for the creation of decentralized financial services and applications. All the different components of Katallassos allow it to offer a better experience for both users and developers than the one that would be possible using current general-purpose blockchains.

We feel confident that Katallassos will help revitalize the current decentralized finance industry by making it simpler than ever to create non-custodial trustless interoperable financial contracts.

VII. Acknowledgments

We would like to acknowledge all other members of the Trinkler Software team, without whom Katallassos would not be possible. In alphabetical order: Addison Huegel, Arie Levy-Cohen, Hervé Fulchiron, Mark Greenslade, Nils Bundi and Seraya Takahashi.

References

[1] B. França, M. Wissfeld, P. Berrang, P. von Styp-Rekowsky, and R. Trinkler, "Albatross: An optimistic consensus algorithm," *White Paper*, 2019. [Online]. Available: https://katallassos.com/papers/Albatross.pdf

[2] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.6130

[3] B. França, "Enso: A general-purpose virtual machine," *White Paper*, 2019. [Online]. Available: https://katallassos.com/papers/Enso.pdf

[4] N. Bundi, "Actus: The algorithmic representation of financial contracts," *White Paper*, 2018. [Online]. Available: https://docs.wixstatic.com/ugd/3df5e2_eceb16e5f7f14d11903a6412aebb9e4a.pdf

[5] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 270–282. [Online]. Available: https://eprint.iacr.org/2016/168.pdf

[6] S. Ellis, A. Juels, and S. Nazarov, "Chainlink: A decentralized oracle network," *White Paper*, 2017. [Online]. Available: https://link.smartcontract.com/whitepaper

[7] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," *White Paper*, 2016. [Online]. Available: https://polkadot.network/PolkaDotPaper.pdf

[8] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, 2014. [Online]. Available: https://cryptorating.eu/whitepapers/Ethereum/Ethereum_white_paper.pdf