Æternity blockchain

The trustless, decentralized and purely functional oracle machine

DRAFT

December 28, 2016

Zackary Hess
zack@aeternity.com

Yanislav Malahov
yani@aeternity.com

Jack Pettersson
jack@aeternity.com

***Abstract*— Since the introduction of Ethereum in 2014 there has been great interest in decentralized trustless applications (smart contracts). Consequently, many have tried to implement applications with real world data on top of a blockchain. We believe that storing the application's state and code on-chain is wrong for several reasons.**

We present a highly scalable blockchain architecture with a consensus mechanism which is also used to check the oracle. This makes the oracle very efficient (cheap), because it avoids layering one consensus mechanism on top of another. State channels are integrated to increase privacy and scalability. Tokens in channels can be transferred using purely functional smart contracts that can access oracle answers.By not storing contract code or state on-chain, we are able to make smart contracts easier to analyze and faster to process, with no substantial loss in *de facto* functionality.

Applications like markets for synthetic assets and prediction markets can be efficiently implemented at global scale. Several parts have proof-of-concept implementations in Erlang. Development tools and application essentials such as a wallet, naming and identity system will also be provided.

CONTENTS

I. INTRODUCTION

The intention of this paper is to give a overview of the Æternity blockchain architecture and possible applications. More detailed papers will be released in the future, specifically for the consensus and governance mechanisms. However, it should be noted that our architecture is holistic; all components tie together and synergize, in a modular way.

The rest of this paper is broken into four parts. First, we will introduce and discuss the fundamental theoretical ideas that inform our architecture. Second, we will discuss the

included essential applications, other possible use cases and give intuitions for how to use the platform as a developer. Third, we will present the current proof-of-concept implementation, written in Erlang. We conclude with a discussion, including possible future directions and comparisons to other technologies.

A. Previous Work

Blockchains, first of all Bitcoin, have shown a new way to architect value exchange on the Internet [1]. This has been followed by a number of promising advances: Ethereum demonstrated a way to write Turing-complete smart contracts secured by a blockchain architecture [2]; Truthcoin created tools for making oracles on blockchains [3], while GroupGnosis and Augur showed how to make them more efficient [4]; Casey Detrio showed how to make markets on blockchains [5]; Namecoin showed how to make the distributed equivalent of a domain name server [6]; Factom showed how a blockchain that stores hashes can be used as a proof of existence for any digital data [7].

These technologies show great promise when it comes to providing first-class financial and legal services to anyone. So far however, they have failed to come together to a unified whole that actually fulfills the promise. Specifically, all solutions so far have been lacking in at least one of the following respects: governance, scalability, scripting safety and cheap access to real-world data [need cit.]. Æternity aims to improve the state of the art in all of these respects.

II. ÆTERNITY BLOCKCHAIN

We believe that the lack of scalability, scripting safety and cheap access to real-world data of current "smart contract platforms" come down to three core issues. *First*, the currently prevailing stateful design makes smart contracts written for the platform hard to analyze[1], and statefulness combined with sequential transaction ordering complicates scalability [need cit.]. *Second*, the high cost of bringing real-world data into the system in a decentralized, trustless and reliable way complicates or outright prevents the realization of many promising applications [need cit.]. *Third*, the platforms are limited in their abilities to update themselves, in order to adapt to new technological or economical knowledge. We believe that each of these three problems have clear solution paths that should be explored.

First, recent research into state channel technology suggests that for many use cases, keeping state on-chain is not necessary [need cit.]. It is very often entirely possible to store all information in state channels, and only use the blockchain to settle any economic results of the information exchange, and as a fallback in the case of dispute. This suggests an alternative approach to blockchain architecture in which Turing-complete smart contracts exist in state channels but not on-chain. This increases scalability since all transactions become independent and can thus be processed in parallel. Additionally, this means that contracts never write to shared state, greatly simplifying their testing and verification. We believe that this design emphasizes that blockchains are about financial logic rather than data storage; there exist decentralized storage solutions that complement blockchains perfectly.

Second, applications such as Augur have attempted to bring real-world data onto the blockchain in a decentralized way—in the process essentially building a consensus mechanism inside smart contracts [8], instead of utilizing the consensus mechanism of the underlying blockchain. This leads to inefficiencies but doesn't increase security. The natural conclusion from this is to generalize the blockchain's consensus mechanism so that it can provide information not only on the next internal state, but also on the state of the external world. It could thus be argued that the blockchain's consensus mechanism determines the result of running what complexity theory dubs an *oracle machine*: a theoretical machine that is more powerful than a Turing machine because it has answers to some questions that can't necessarily be computed, such as "Who won football game X?" [need cit.].

Third, it seems natural that the consensus mechanism could also be used to determine the parameters of the system. This allows it to adapt to changing external conditions, as well as adopting new research and recent developments in the field.

The rest of this section introduces the Æternity blockchain in greater detail, starting with a brief overview of accounts, tokens, names and the structure of blocks. This is followed by an explanation of our approach to state channels and smart contracts, and then a discussion on how the blockchain's consensus mechanism can be used both to create an efficient oracle mechanism and to govern the system. Finally, we discuss scalability from several different angles.

A. Tokens, accounts and blocks

Despite being "stateless" from the contract developer's point of view, the Æternity blockchain keeps track of several predefined state components. We will now explain these, as well as the content of each block. For simplicity, this section assumes that every node keeps track of the whole blockchain. Possible optimizations are described in section II-E.

A.1) Access token, Aeon: To use the blockchain is not free, but requires that the user spends a token called *aeon*. Aeon are used as payment for any resources one consumes on the platform, as well as the basis for financial applications implemented on the platform.

The distribution of aeon in the genesis block will be determined by a smart contract hosted on Ethereum. Further aeon will be created via mining.

All system fees get paid with aeon, all smart contracts settle in aeon.

A.2) Accounts: Each account has an address and a balance of aeon and also a nonce which increases with every transaction and the height of its last update. Each account

[1]The difficulty of analyzing stateful contracts was very clearly demonstrated by the re-entrance bug that brought down "The DAO". This happened despite the code having been audited by several of Ethereum's creators as well as the general community [need cit.].

also has to pay a small fee for the amount of time it is open. The costs of creating and keeping accounts prevents spam and disincentivizes state-bloat. The reward for deleting accounts incentivizes the reclaiming of space.

A.3) Name system: Many blockchain systems suffer from unreadable addresses for their users. In the vein of Aaron Swartz' work and Namecoin, Æternity features a name system that is both decentralized and secure, while still supporting human-friendly names [9]. The blockchain's state includes a mapping from unique human-friendly strings to fixed-size byte arrays. These names can be used to point to things such as account addresses on Æternity, or hashes e.g. of Merkle trees.

A.4) Block contents: Each block contains the following components:

- The hash of the previous block.
- A Merkle tree of transactions.
- A Merkle tree of accounts.
- A Merkle tree of names.
- A Merkle tree of open channels.
- A Merkle tree of oracles which haven't answered their respective questions.
- A Merkle tree of oracle answers.
- A Merkle tree of Merkle proofs.
- The current entropy in the random number generator.

The hash of the previous block is required to maintain an ordering of the blockchain. The transaction tree contains all transactions that are included in the current block. With the exception of the consensus vote tree, all the trees are fully under consensus: if a tree is changed from one block to the next, this change has to be due to a transaction in the new block's transaction tree, and a Merkle proof of the update has to be included in the block's proof tree. The purpose of the three remaining trees will hopefully become clear in the following sections.

B. State channels

One of the most interesting developments in the blockchain space lately is that of state channels. They operate on the basic principle that in most cases, only the people affected by a transaction need to know about it. In essence, the transacting parties instantiate some state on a blockchain, e.g. an Ethereum contract or a Bitcoin multisig. They then simply send signed updates to this state between each other. The key point is that either one of them could use these to update the state on the blockchain, but in most cases, they don't. This allows for transactions to be conducted as fast as information can be transmitted and processed by the parties, instead of them having to wait until the transaction has been validated—and potentially finalized—by the blockchain's consensus mechanism.

On Æternity, the *only* state update that can be settled on the blockchain is a transfer of aeon, and the only aeon that can be transferred are the ones that the transacting parties already deposited into the channel. This makes all channels independent from each other, which has the immediate benefit that any transactions related to channels can be processed in parallel, greatly improving transaction throughput.

```
1 macro Gold f870e8f615b386aad5b953fe089 ;
2
3 Gold oracle
4 if 0 1000 else 0 0 end
5 0
```

Fig. 1. A simple contract encoding a bet on the price of gold. The language used is the Forth-like *Chalang*, which will be presented in section IV-A.

The blockchain is only used to settle the final outcome or to resolve conflicts that arise, roughly analogous to the judicial system. However, because the blockchain's behavior will be predictable, there is no gain in disputing the intended result of a state channel; malicious actors are incentivized to behave correctly and only settle the final state on the blockchain. All taken together, this increases transaction speed and volume by several orders of magnitude, as well as privacy.

B.1) Smart contracts: Despite that the only state that can be settled on-chain is a transfer of aeon, Æternity still features a Turing-complete virtual machine that can run "smart contracts". Contracts on Æternity are strictly agreements that distribute funds according to some rules, which stands in stark contrast to the entity-like contracts of e.g. Ethereum. Two of the more notable practical differences is that by default, only the involved parties know about a given contract, and only parties that have an open state channel can create a valid contract. If the parties agree to a contract, they sign it and keep copies for future reference. It is only submitted to the blockchain if its outcome is disputed, in which case the code is only ever stored as part of the submitted transaction, never in any other state. If this happens, the blockchain distributes the tokens according to the contract and closes the channel.

As an example, fig. 1 shows a very simple contract that encodes a bet on the price of gold at a certain time. On line 1, the macro `Gold` saves the identifier of the oracle in question, which will return `true` if the price of gold is below $38/g on December 1st, 2016. The body of the contract is displayed on lines 2-4: we first push the gold oracle's identifier to the stack and call it using `oracle`, which will leave the oracle's answer on the top of the stack. We use this to do a conditional branching: if the oracle returns `true`, we push 0 and 1000 to the stack, indicating that 0 aeon should be burned and 1000 aeon should go to the first participant in the channel. Otherwise, we push 0 and 0, with the second 0 indicating that the other participant receives all aeon in the channel. Finally we push 0, which is taken to be the nonce of this channel state. In actual usage, the nonce would be generated at deployment.

One important thing to note is that contracts on Æternity don't maintain any state of their own. Any state is maintained by the transacting parties and submitted as input at execution. Every contract is essentially a *pure function* that takes some

```
1 : hashlock
2 swap
3 hash
4 == ;
```

Fig. 2. A simple hashlock.

```
1 macro Commitment a9d7e8023f80ac8928334 ;
2
3 Commitment hashlock call
4 if 0 100 else 0 50 end
5 1
```

Fig. 3. Using the hashlock to trustlessly send tokens through a middleman.

input and gives a new channel state as output[2]. The benefits of using pure functions in software development in general, and in the development of financial applications in particular, has been extensively documented in academia and industry for decades [10][need cit.].

a) Contract interaction and multi-step contracts: Even though all contracts are stateless and execute independently of each other, contract interaction and statefulness can still be achieved through *hashlocking* [need cit.]. A simple hashlock is shown in fig. 2. On line 1, we define a function called `hashlock` that expects the stack to contain a hash h and a secret s. It swaps them on line 2, in order to hash the secret on line 3, before calling the equality operator on $hash(v)$ and h on line 4. This returns true if the secret is a preimage of the hash. This function can be used to predicate the execution of code branches in different contracts on the existence of the same secret value.

As a simple example usage, hashlocks make it possible for users that don't share a state channel to trustlessly send each other aeon, as long as there is a path of channels between them. For example, if Alice and Bob have a channel and Bob and Carol have a channel, then Alice and Carol can transact through Bob. They do this by creating two copies of the contract shown in fig. 3, one for each channel. The `Commitment` on line 1 is the hash of a secret that Alice chooses. On line 3 we push it to the stack and call the `hashlock` function. Which branch of the `if` that gets executed depends on the return value of `hashlock`. Once these contracts have been signed by all parties, Alice reveals the secret, allowing Bob and Carol to use it to claim their aeon.

Hashlocking can also be used to e.g. play multi-player games in the channels, as shown in fig. 4. Everyone makes a channel with the game manager, which publishes the same contract to every channel. Say we are in game state 32,

[2]It should be noted that since contracts can read answers from oracles and some environment parameters, they aren't completely pure functions. However, oracle answers never change once they've been provided and can be argued to be due to the computational richness of the oracle machine, rather than being an impurity. Environment parameters are deemed a "necessary evil" and will ideally be compartmentalized appropriately by high-level languages.

```
1 macro Commitment a9d7e8023f80ac8928334 ;
2
3 Commitment hashlock call
4 if State33 else State32 end
5 call
```

Fig. 4. A simplified example of using the hashlock to play a multi-player game in channels.

defined by the function `State32`, and we want to trustlessly simultaneously update all the channels to state 33. When the game manager reveals the secret, it causes all the channels to update at the same time.

b) Metered execution: Contract execution is metered in a way similar to Ethereum's "gas", but Æternity uses two different resources for its metering, one for time and one for space. Both of these are paid for using aeon by the party that requests the execution.

This could be seen as undesirable, because it is probably another party that is causing the need for the blockchain to resolve the dispute in the first place. However, as long as all money in the channel is not used for betting, this can be effectively nullified in the contract code, since it has the ability to redistribute funds from one party to the other. It is in fact generally good practice to avoid using all funds in a channel to transact, because it disincentivizes the losing party to cooperate when closing the channel.

B.2) Example: Let's bring all of these ideas down to earth. In practice, if Alice and Bob want to transact using a state channel on Æternity, they go through the following procedure:

1) Alice and Bob sign a transaction that specifies how much money each of them is depositing into the channel, and publish it to the blockchain.
2) Once the blockchain has opened the channel, they can both create new channel states, send them between each other and sign them. Channel states can be either a new distribution of the funds in the channel or a contract that determines a new distribution. Each of these channel states has an increasing nonce and are signed by both parties, so if a dispute arises, the latest valid state can be submitted to the blockchain, which enforces it.
3) The channel can be closed in two different ways:
   a) If Alice and Bob decide that they have finished transacting and agree on their final balances, they both sign a transaction indicating this and submit it to the blockchain, which will close the channel and redistribute the money in the channel accordingly.
   b) If Alice refuses to sign a closing transaction for any reason, Bob can submit the last state that both of them signed and request to have the channel closed using this state. This starts a countdown. If Alice believes that Bob is being dishonest, she has the opportunity to publish a state with

a higher nonce that both of them have signed before the countdown finishes. If she does so, the channel closes immediately. Otherwise it closes when the countdown has finished.

C. Consensus mechanism

Since most computation will happen in the channels rather than on-chain, we will use a slow-finality consensus mechanism. This makes the consensus more affordable. The consensus mechanism will use a small amount of proof-of-work, so that the number of possible forks will stay small and can fit in everyone's memory. Deciding which fork to use will be heavily influenced by the outcomes of prediction markets, which tell us which decisions are best. Proposing and analyzing our mechanism is outside the scope of this paper and will be done in an imminent complementary paper.

Preview: The consensus mechanism has a somewhat non-standard role in Æternity. In addition to agreeing on new blocks for the blockchain, it also agrees on both answers to oracle questions and the values of the system's parameters. In particular, the consensus mechanism can change itself. However, it should be noted that this is not entirely unproblematic. For example, if a simple proof-of-work mechanism was used, it would be rather cheap to bribe the miners to corrupt the oracle. Therefore Æternity is going to use a novel Proof-of-Stake algorithm, with a smaller Proof-of-Work addition, leveraging the benefits of both. Independently from this, PoW is going to be used to issue new aeon.

C.1) Oracles: A crucial feature for most contracts, whether encoded as text or as code, is the ability to refer to values from the environment, such as the prices of different goods or whether a certain event occurred or not. A smart contract system without this ability is essentially a closed system and arguably not very useful. This is a generally accepted fact and there are already several projects that attempt to bring external data into the blockchain in decentralized way [8]. However, to decide whether a supplied fact is true or not, these essentially require the implementation of a new consensus mechanism on top of the consensus mechanism.

Running two consensus mechanisms on top of each other is as expensive as running both separately. Additionally, it doesn't increase security, because the least secure one can still be attacked and made to produce "false" values. Thus, we propose to conflate the two consensus mechanisms into one, essentially reusing the mechanism that we use to agree on the state of the system, to also agree on the state of the outside world.

The way that this works is as follows. Any aeon-holder can launch an oracle by committing to answering a yes/no-question. When doing so, they also need to specify the timeframe during which the question can be answered, which can start now or some time in the future. The user that launches the oracle is required to deposit aeon in proportion to the length of the timeframe, which will be returned if the user supplies an answer that gets accepted as truth, otherwise it is burned. The blockchain generates a unique identifier for the oracle that can be used to retrieve the answer once it is available.

Once the time comes for the question to be answered, the user who launched the oracle can supply an answer for free. Once the oracle launcher has supplied an answer or until a certain amount of time has passed, any other users can submit counter-claims by depositing the same amount of aeon. If no counter-claims have been submitted by the end of the timeframe, the answer supplied by the user that launched the oracle is accepted as truth, and the deposit is returned. If any counter-claims are submitted, then the consensus mechanism for blocks will be used to answer the oracle. This is more expensive, but since we know we can take at least one of the two safety deposits, we can use it.

D. Governance

Governance of blockchain-based systems has been a big problem in the past. Whenever a system upgrade needs to be done, this requires a hard fork, which usually leads to big discussions among all value holders. Even simple things, like correcting an arbitrarily set variable in the source code, as we have seen with the block size debate in Bitcoin, seem to be very hard in a system where the users' incentives are not aligned with the decision makers, and where there is no clear upgrade path. We have also seen more complicated governance decisions, like fixing a single smart contract bug in "The DAO", which required quick intervention by system developers.

The primary problem of these systems is easily identifiable—the decision-making process for a protocol upgrade or change is not well defined and lacks transparency. Æternity's governance system is part of the consensus. It uses prediction markets to function as efficiently and transparently as possible.

Moreover, the consensus mechanism is defined by a number of variables that determine how the system functions and that are being slightly updated by each new block. From how much it costs to make transactions or ask an oracle, to modifications of fundamental parameter values like the block time.

By having prediction markets about the variables that define the protocol, the users can learn how to efficiently improve the protocol. By having predictions markets about potential hard forks, we can help the community come to consensus about which version of the code to use. Each user chooses for itself which metric it seeks to optimize, but a simple default strategy would be to maximize the value of its holdings.

E. Scalability

E.1) Sharding trees: The architecture that has been presented thus far is highly scalable. It is possible to run the blockchain even when each user only keeps track of the part of the blockchain state that they care about and ignores everyone else's data. At least one copy of the state is needed for new users to be certain about the substate that they care about, but we can shard this data across arbitrarily many

nodes so that each node's load is arbitrarily small. Merkle trees are used to prove that a substate is part of the state [11]. It is easy to imagine a scenario where certain nodes specialize on keeping track of the trees and get paid for inserts and look-ups.

E.2) Light clients: Light clients don't download the entire blocks. First the user gives their client a hash in the history of the fork they prefer, a technique also known as *weak subjectivity* [12]. Then the client knows only to download forks that include a block with that hash. The client only downloads the headers of the blocks. The headers are much smaller than full blocks; very few transactions are processed. For simplicity, we made no mention of the block headers when discussing the block structure in section II-A.4, but they contain the following:

- The hash of the previous block.
- The root hash of all of the state trees.

E.3) State channels and parallelism: State channels have immense throughput and most transactions inside them are never executed or even recorded on the blockchain. Additionally, the channels don't write to any shared state on-chain, so all transactions that actually *do* get recorded on the blockchain can be processed in parallel. Given that most consumer hardware sold today has at least four processing cores, this has the immediate effect that transaction throughput is multiplied by roughly a factor of 4.

Furthermore, the fact that there will never be any complex concurrent interaction suggests that sharding this blockchain architecture should be relatively easy. Since blockchain sharding is still fairly experimental, we have deliberately chosen not to pursue any sharding techniques in the initial design of Æternity. However, if this changes in the future, Æternity should be one of the easiest blockchains to shard.

E.4) Transactions per second at a given memory requirement: The variables that define the protocol are all constantly being updated by the consensus. From their initial default values, we can calculate the initial default rate of transactions per second.

```
Note that this is a draft and will likely
change.

We define the following variables for the
    following calculations:

B = block\_size in bytes
F = blocks\_till\_finality
R = time\_till\_finality in seconds
T = transaction size in bytes

transactions per second = B * F / (T * R)

B = 1000000 bytes = 1 megabyte per block
F =  24*60*2 blocks per day
R / F = 30 seconds per block
R = 24*3600 seconds per day
T = 1000 bytes per transaction

1000000 * 24*60*2 / 1000 / 24*3600
= 1000000 / 1000 / 30
= ca. 32 transactions per second (fast
    enough to sign up every human within 8
    years)
```

To operate a node, we need to keep a copy of all the blocks since finality, and we need to be able to record 100 times more information, in case there is an attack. Estimating that finality is 2 days, then there would be 5760 blocks till finality. So the memory requirement is 5760 * one megabyte * 100 = 576000 megabytes = 576 gigabytes. When there isn't an attack happening, one would only need about 5.76 gigabytes to store the blocks.

III. Applications

The stateless nature of the Æternity smart contracts makes it easy to build the following applications on Æternity's blockchain. It is especially suitable for high-volume use-cases.

A. Blockchain essentials

Blockchain essentials are necessary primitives like aeon, wallets, names and related concepts. They modularize reusable components which can be used as application foundations and can be improved on.

A.1) Identities: Each account will have an associated unique ID number. Users can register unique names, and link names to the Merkle-root of a data structure. The data structure can contain one's unique ID as well as other information about one's account. We aim to use Schema.org's JSON format to represent things like persons or companies [13].

A.2) Wallet: A wallet is a piece of software that is used to interact with Aeternity. A wallet manages private keys for the aeon and creates and signs transactions. One can use the wallet to send channel transactions, and use apps in the channel network.

A.3) Proof of existence: One transaction type allows for the publishing of the hash of any data. System participants can use the headers to prove that the data existed at that point in time.

B. State channel applications

Smart contracts in state channels are perfect for micro-services on the web that require a high transaction throughput.

B.1) Toll API: Most APIs existing today are publicly available for anyone to call, or else they are secured by a username-password–scheme or unique access tokens. Payment channels allow for a new kind of API, where one pays for every call to the API, possibly every HTTP-request. Paying to access an API solves DDoS problems, and it makes it easier to build high-quality APIs that are always available. API responses that require a payment are fundamental for the creation of as of yet impossible types of businesses and can play an important role in the emergence of the decentralized economy. They create incentives for information owners to make otherwise private data publicly available.

B.2) Insured crowdfunding: We can implement insured crowdfunding using dominant assurance contracts [need cit.]. These are smart contracts that are used to raise money for a public good, like a new bridge, a school or a market.

Dominant assurance contracts differ from traditional assurance contracts like Kickstarter, in that they make it a dominant strategy to participate. If the good is not funded, all participants get their aeon back plus interest, so they are insured against reducing their liquidity without receiving the good. Using an oracle, we can ensure that the provider of the good or service only gets paid if the good or service is actually provided.

B.3) Cross-chain atomic swaps: Cross chain atomic swaps allow for trustless exchange of aeon for bitcoins [14], [15]. These can be implemented using a hashlock, that locks the transactions on both blockchains under the same value.

B.4) Stable value assets and portfolio replication: We can use smart contracts to program synthetic assets that stay nearly the same price as a real world asset. For example, we could make an asset that stays the same price as gold. Synthetic derivatives are created in equal and opposite pairs. For one user to have an asset that moves with gold, a different user will have to have an asset that move inversely to gold. For example, Alice could make a contract with Bob so that Alice owns 1 gram of gold. Out of the money in the contract, one gram of gold worth of aeon will go to Alice, and the leftover money goes to Bob. The contract has an expiration date when the price of gold will be measured, and the funds distributed to Alice and Bob accordingly.

B.5) Event contracts: Event contracts pay when an event happens and don't pay when an event does not happen, as per the oracle's telling. Apart from being interesting in themselves, these can be used by several different applications:

a) Insurances: We can use event contracts to implement insurances. For example, expensive music event tickets can become worthless if the weather goes bad. However, if the concert-goer receives money if the oracle decides that it rained on the day of the event, the investment can be protected so that one can afford to find an emotionally-adequate alternative. Slightly more seriously, farmers are often interested in the total number of inches of rain in a season. We can insure them against their crops wilting from dryness.

b) Whistleblowing: Event contracts can also be used to incentivize revealing sensitive information. For example, we could bet on the event "Information indicating that Company A has used illegal pesticides was released on or before January 24th, 2017". Any person with access to such information would be incentivized to first bet that the event will happen and then release it.

B.6) Prediction markets: A prediction market works by letting users bet on whether a future event will happen. From the price of the bets we can predict the future likelihood [3], [8], [16]. They are the most accurate way to measure the future at a given price [need cit.]. Once the event has happened, the market is settled using the oracle.

As noted in section II-D, we can for example use prediction markets to predict which updates to the software will be beneficial, and which will be harmful. We can also use them to estimate how much candidates in an election will actually be able to accomplish, so lies and baseless promises can be detected more easily.

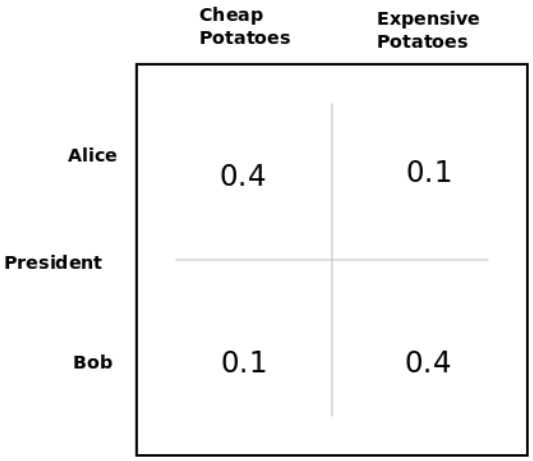

Fig. 5. Multidimensional prediction market.

a) Multidimensional prediction markets: Multidimentional prediction markets allow us to predict the correlation between possible future events. So for example, one could predict that if Alice is elected leader, the price of potatoes will go down, and that if Bob wins, the price will go up. One could learn that if Google uses plan A for the next 3 months, that it will probably earn more money, and that if it uses plan B, it will probably earn less. Or, as in fig. 5, we can see that if Alice would be elected president, there is a high likelihood of the price of potatoes being rather low.

B.7) Market with batch trading at a single price: There are two approaches available to attackers that want to rob aeon from a market. They can take advantage of the market being split in time, or they can take advantage of it being split in space.

- If the market is split in space, then the attacker does arbitrage. He simultaneously makes trades in both markets at once so that his risk cancels out and he earns a profit.
- If the market is split in time, then the attacker front-runs the market. He reads the transactions coming into the market and creates buy and sell orders immediately before and after.

To combine markets in space, everyone should use the same market maker. To combine markets in time, we need to have trading done in batches, at single price. The market maker needs to commit to each person what price he decided, and if anyone can find contradictory commitments from the market maker, then all of his customers should be able to drain all of his channels. If the market maker commits to a fair price, then he will match the same volume of buyers and sellers together, as fig. 6 shows. Otherwise, he will end up in a situation similar to fig. 7, thus taking a large risk.

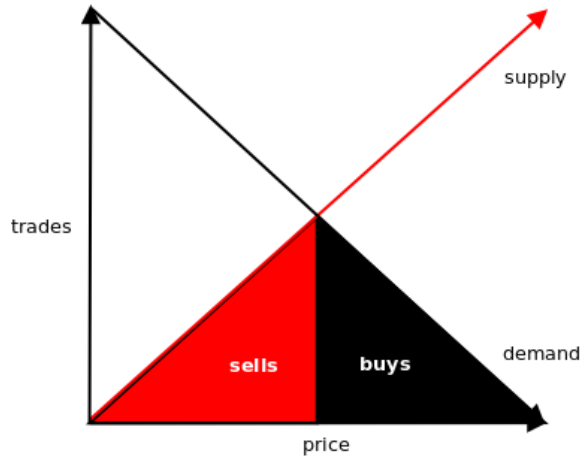

Fig. 6. The black line is the demand curve, the red line is the supply curve. The sells in red are the same size as the buys in red. The vertical line is the price the market maker selected. Everyone willing to buy at a higher price traded at that price, everyone willing to sell at a lower price traded at that price.

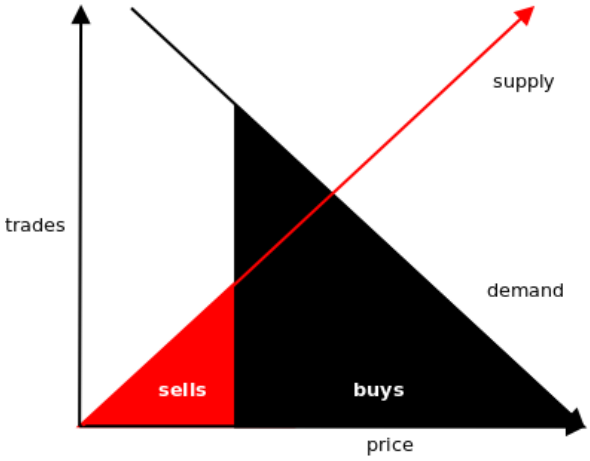

Fig. 7. The black is much bigger than the red. The market maker is selling many more shares than it is buying, thus taking on a lot of risk.

IV. Implementation

Most key concepts already have proof-of-concept implementations in Erlang. This includes the blockchain itself, the contract language and VM, the oracle and governance mechanisms, as well as an old version of the consensus mechanism. We have used Erlang/OTP because it makes it easy to write code that can respond to many requests in parallel and does not crash. The servers with the highest uptime in the world are based on Erlang. It has been used for industrial applications for 30 years, proving itself to be a reliable and stable product.

A. Virtual machine and contract language

The virtual machine is stack-based and similar to Forth and Bitcoin' scripting language, although in comparison to the latter, it is rather rich. The VM supports functions instead of gotos, making its semantics relatively simple to analyze. A list of the VM's opcodes can be found on our Github[3].

Additionally, there exists a higher-level Forth-like language called *Chalang*, which compiles to bytecode for the VM. It supports macros and variable names, but keeps the stack-based execution model [17]. Examples of Chalang code can also be found on our Github[4].

B. Adoption via web-integration

The web is the most popular application platform. We will provide easy-to-use web-development tools, such as JS-libraries and JSON-APIs for the core features of the Æternity blockchain.

C. Open source modules

In order to be easily re-used for private blockchain consortium and other use-cases, the software will be written in MIT-licensed modules, such as a consensus module, that can be adapted to specific needs.

D. Usability and UX design

Frictionless human interaction will be a big focus of our development efforts. More specifically, we will make sure that who controls the identity, keys and transactions is clearly established. Also, offering easy access via web-gateways will be a central focus of future development. Users participating in prediction markets via a Tinder-like (swipe left/right) mobile interface, and simple web-wallets that can be easily integrated in a website through an iframe will be the new norm.

V. Discussion

We have provided an explanation of how to architect a fundamentally more efficient value transfer system. The described system is in fact a global oracle machine that can be used to provide decision making services at global scale. In particular, all the applications proposed in section III can be built easily and efficiently on top of Æternity.

However, our approach has both fundamental limitations and avenues for improvement. These are discussed here.

A. Limitations and tradeoffs

While we do believe that the tradeoffs made in our architecture are reasonable given the resulting performance increase in other areas, Æternity is not a catch-all solution for decentralized applications. It should rather be viewed as a synergistic complement to existing technologies. There are several caveats that one need to be aware of.

A.1) On-chain state: Despite having many advantages, Æternity's lack of programmable state makes it unfit for applications that require a custom state to be under consensus. For example, this includes DAOs as they are usually conceived, custom name systems and subcurrencies which are not tied to the value of an underlying asset.

[3] https://github.com/aeternity/chalang/blob/master/opcodes.md

[4] https://github.com/aeternity/chalang/tree/master/examples

A.2) Free option problem: If Alice and Bob have a channel and Alice signs a contract, she essentially gives Bob a free option when she sends it to him: Bob can choose to sign and return (i.e. activate) the contract at any time in the future. Often this is not what is intended. To avoid this problem, channel contracts aren't immediately activated with the full amount. They are divided up in time or space. Both participants would sign up for the contract in small intervals so that neither user ever offers a large free option to the other.

For example, if the parties want to bet 100 aeon, then they might sign up to it in 1000 steps that each increase the bet by 0.1 aeon. This would require about 1000 messages to pass, 500 in each direction, which is cheap enough since the contract is never submitted to the blockchain. As another example, if one wanted to make a financial asset that would last for 100 days, one might sign up in 2400 steps of one hour each. This would require about 2400 messages to pass, 1200 in each direction.

A.3) Liquidity loss and state channel topologies: When composing channels using hashlocks as demonstrated in section II-B.1, any middlemen have to lock up at least twice as many aeon as will be transmitted through them. For example, if Alice and Carol want to transact through Bob, Bob will act as Carol when interacting with Alice, and vice-versa.

Since this is expensive for Bob, he would most likely earn a fee as compensation. If Alice and Carol expect to conduct many trades between each other, they can avoid this by creating a new channel and trustlessly moving the active contracts to the new channel using a hashlock.

Still, since keeping an extra channel open impacts one's liquidity negatively, going through middlemen is expected to be desirable in many cases, especially in cases where the parties don't expect to trade a lot in the future. Thus, a channel topology where certain rich users make money from trustlessly transmitting transactions between other users is expected to emerge.

It should be noted that this does not constitute a single point of failure, since we do not trust these transaction transmitters with anything. If a transmitter goes offline before the secret to a hashlock has been revealed, the transaction doesn't go through. If it goes offline afterwards, the only possible "negative" effect is that the transmitter is not able to claim its aeon.

B. Future work

There are several possible ways to improve on the current architecture.

B.1) Functional contract language: A reasonable future direction would be to experiment with high-level languages that adhere more closely to the functional paradigm. Keeping track of an implicit stack is generally error-prone and arguably not suitable for a high-level, developer-facing language. This should be rather easy given that programs are already pure functions (modulo some environment variables), and would greatly simplify both development and formal verification of contracts. If this is done, it could also make sense to revise the VM to be tightly coupled with the new language, to make the compilation less error-prone and less dependent on trust in the developers. Ideally, the translation from surface language to VM code would simply be a direct transcription of peer-reviewed research, though pragmatic concessions will likely have to be made.

B.2) Multi-party channels: Currently, all channels on Æternity are limited to two parties. While multi-party channels can *de facto* be achieved through hashlocking, this can be expensive. Hence, we plan to investigate the possibility of adding support for n-party channels, with a m-of-n settlement mechanism.

GLOSSARY

Blockchain A distributed, tamper-proof database with metered access. The database is defined by a growing list of hash-linked blocks and can have any rules for appending them.

Aeon An aeon represents an unit of account and an access right to the Æternity blockchain. It is transferable.

Transaction A message from a user to the blockchain. This is how users can use their currency to access the blockchain.

State Channel A relationship between two users recorded on the blockchain. It enables users to send aeon back and forth, and to create trustless smart contracts between them that are enforced and settled by the blockchain.

Hash A hash takes as input a binary of any size. It gives a fixed sized output. The same input always hashes to the same output. Given an output, one cannot calculate the input.

Hashlocking This is how we connect pairs of channels to make smart contracts that involve more than 2 people. A secret is referenced by it's hash. When the secret is revealed, it can update multiple channels at the same time.

Governance A well-defined process of making decisions for the future protocol(s) of the blockchain.

Oracle A mechanism that tells the blockchain facts about the world we live in. Using oracles users can predict the outcome of events, external to the blockchain system.

Value-Holder An user who owns aeon, or an financial derivative in the system.

Validator A validator is an user who participates in the consensus mechanism. In the case of Æternity, every value-holder can participate.

ACKNOWLEDGMENTS

Thanks to Vlad, Matt, Paul, Dirk, Martin, Alistair, Devon and Ben for proof-reading. Thanks to these and lots of other people for insightful discussions.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: `https://bitcoin.org/bitcoin.pdf`.

[2] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper.

[3] P. Sztorc, "Market empiricism," [Online]. Available: http://bitcoinhivemind.com/papers/1_Purpose.pdf.

[4] M. Liston and M. Köppelmann, "A visit to the oracle," 2016. [Online]. Available: https://blog.gnosis.pm.

[5] C. Detrio, "Smart markets for smart contracts," 2015. [Online]. Available: http://cdetr.io/smart-markets/.

[6] *Namecoin wiki*, 2016. [Online]. Available: https://wiki.namecoin.org/index.php?title=Welcome.

[7] P. Snow, B. Deery, J. Lu, *et al.*, "Factom: Business processes secured by immutable audit trails on the blockchain," 2014. [Online]. Available: http://bravenewcoin.com/assets/Whitepapers/Factom-Whitepaper.pdf.

[8] J. Peterson and J. Krug, "Augur: A decentralized, open-source platform for prediction markets," 2014. [Online]. Available: http://bravenewcoin.com/assets/Whitepapers/Augur-A-Decentralized-Open-Source-Platform-for-Prediction-Markets.pdf.

[9] A. Swartz, "Squaring the triangle: Secure, decentralized, human-readable names," 2011. [Online]. Available: http://www.aaronsw.com/weblog/squarezooko.

[10] T. Hvitved, "A Survey of Formal Languages for Contracts," in *Formal Languages and Analysis of Contract-Oriented Software*, 2010, pp. 29–32. [Online]. Available: http://www.diku.dk/hjemmesider/ansatte/hvitved/publications/hvitved10flacosb.pdf.

[11] R. C. Merkle, "Protocols for public key cryptosystems," in *IEEE Symposium on Security and Privacy*, 1980.

[12] V. Buterin, "Proof of stake: How I learned to love weak subjectivity," 2014. [Online]. Available: https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity/.

[13] "Schema.org schemas," 2016. [Online]. Available: http://schema.org/docs/schemas.html.

[14] "Atomic-cross-chain-trading," 2016. [Online]. Available: https://en.bitcoin.it/wiki/Atomic%5C_cross-chain%5C_trading.

[15] "Interledger," 2016. [Online]. Available: https://interledger.org/.

[16] K. J. Arrow, R. Forsythe, M. Gorham, *et al.*, "The promise of prediction markets," *Science*, 320 2008. [Online]. Available: http://mason.gmu.edu/~rhanson/PromisePredMkt.pdf.

[17] Z. Hess, "Chalang," 2016. [Online]. Available: https://github.com/aeternity/chalang.