AALBORG UNIVERSITY

DENMARK

SmartExchange

Decentralised Trustless Cryptocurrency Exchange

Adamik, Filip; Kosta, Sokol

SmartExchange: Decentralised Trustless Cryptocurrency Exchange

Filip Adamik and Sokol Kosta

CMI, Aalborg University Copenhagen, Denmark
`fadami15@student.aau.dk,sok@es.aau.dk`

Abstract. Trading cryptocurrency on current digital exchange platforms is a trust-based process, where the parties involved in the exchange have to fully trust the service provider. As it has been proven several times, this could lead to funds being stolen, either due to malicious service providers that simply disappear or due to hacks that these platforms might suffer. In this work, we propose and develop a decentralised exchange solution based on smart contracts running on the Ethereum network that is open, verifiable, and does not require trust. The platform enables two parties to trade different currencies, limited to Ethereum and Bitcoin in the current status of the system. A smart contract, deployed on the Ethereum blockchain, functions as an *escrow*, which holds a user's funds until a verified transaction has been made by the other party. To make the smart contract able to detect a Bitcoin transfer, we implement our solution by utilising an *oracle*. We define the system architecture and implement a working platform, which we test in a model scenario, successfully exchanging Bitcoin and Ether on the blockchain test networks. We conclude the paper identifying possible challenges and threats to such a system.

Key words: cryptocurrency, distributed, exchange, blockchain, smart contract, oracle, ethereum, bitcoin

1 Introduction

Researchers and developers have proposed cryptocurrency as early as 1983, when anonymous digital money was first introduced [1]. Today's cryptocurrencies, such as Bitcoin, Litecoin, Ethereum, etc., are thus only the continuation of a long lasting effort on developing a currency with sufficient guarantees against misuses, such as double-spending or theft. The wide-spread adoption of these currencies indicates that the guarantees against misuse are indeed sufficient – in other words, that the users trust the cryptocurrencies. If this was not the case, they probably would not be used. There are numerous reasons why users buy cryptocurrency, ranging from value-holding purposes to short-term trading. However, the trust in the cryptocurrency is somewhat unavoidable. Regardless of the motivation, whenever people acquire a cryptocurrency, they believe that it will retain value, at least until they exchange it for other services or goods.

Indeed, exchanging cryptocurrencies has become a regular task for an increasing number of people. There are countless different platforms that offer this service, ranging from simple wallets that embed currency exchange as additional service, to massive trading platforms with daily market cap of millions of dollars, such as *Bitfinex*[1], *Coinbase*[2], or *Bitstamp*[3], among others. All these exchanges, however, share one common feature: when making a transaction, users need to trust them with their funds. For example, whenever a user exchanges USD to Bitcoin, she transfers the USD funds to the company's account, which converts them to Bitcoin. Until the user does not withdraw the converted amount, the platform has full control over her money. If the company suddenly goes out of business, or gets attacked, the user may never get her funds back! This situation is not hypothetical, there have been various cases where deposited funds were lost or stolen, with the most famous case of the Japanese-based Bitcoin exchange Mt. Gox, where USD 380 million worth of Bitcoins were stolen in 2014 [2]. As such, the trust in the exchange platforms is quite questionable. Some experts advise not to store any funds in these systems for long periods of time and to make deposits to a secured wallet right away, since it is believed the funds are more vulnerable in an exchange platform [3].

In this work, we address this problem and propose an exchange system where the trust in the currency is used for making the exchange process of exchange safer. To accomplish this task, we use the Ethereum system, which is the second most popular cryptocurrency as of today. Ethereum gives the possibility of running *smart contracts* – small programs that are run by many computers around the world at the same time. This provides assurance that no-one will tamper with the code and that no-one can manipulate the outcome of a smart contract. We design and implement a system that deploys smart contracts to the Ethereum network, which are used to control the trading process and function. We show that having a distributed exchange platform is not only feasible, but is highly beneficial for the users, since it removes the trust from a single entity. Moreover, to facilitate the utilization of the exchange system, we implement an Android demo application that we use for testing purposes. We demonstrate the operation of this prototype in a model scenario, where we transfer Bitcoin and Ether between two trading parties.

The rest of this work is structured as follows: First, in Section 2 we investigate the existing systems and platforms related to our proposal; Then, in Section 3 we describe the system architecture and the implementation details of our platform and demo application; In Section 4 we present the evaluation of the distributed exchange platform, deploying it on the test blockchain networks and performing Ethereum to Bitcoin currency exchange; In Section 5 we present a short discussion about the limits of our proposal; and finally, in Section 6 we conclude the work presenting the final remarks and considerations for future improvements.

---

[1] `www.bitfinex.com`

[2] `www.coinbase.com`

[3] `www.bitstamp.com`

2 Related Work

Blockchain is a distributed ledger that records sets of *changes* in a system [4]. A *change* could be any data – for example it could be details of the latest currency transactions [5] or newly deposited business agreements [6]. Blockchain is usually distributed among several independent parties to prevent centralisation of the system and to avoid easy manipulation with the blockchain data by a malicious party [7]. The most notable use of the blockchain is the decentralised peer-to-peer cryptocurrency Bitcoin, where the blockchain is used to keep record of all currency transactions among its users [5]. Researchers identify the transaction-recording as the most standard use of the blockchain [7, 8]. However, the possible uses of the blockchain extend further than that – blockchains can be used for operating decentralised applications [9], storing documents [6], financial uses, or outside the financial world [7].

Smart contracts are small programs run in a decentralised manner by all the nodes participating in the network, which are the heart of financial utilization of blockchains. A smart contract consists of a program code, a balance, and a storage space [10, 11]. The most prominent system that operates smart contracts is the Ethereum platform. Ethereum fuses the blockchain with a Turing-complete programming language run by the *Ethereum Virtual Machine (EVM)* [12]. The states of this machine are altered by transactions. Besides invoking functions of a smart contract, an Ethereum transaction can also be used to transfer Ethereum's native cryptocurrency *Ether*. For any transaction to be executed by the EVM, users need to purchase command-execution units called *gas*. Users purchase gas to pay for the operation of the EVM – every smart contract must have enough gas to cover for all of its instructions, otherwise it will not be executed [13].

Researchers identify benefits of using the blockchain beyond the finance sector by providing a decentralised and trusted storage system. There are exemplary uses by IBM and Microsoft, who provide the blockchain as a cloud-based service for storing data and assuring their existence [14], or an e-commerce platform implementation by a china-originated multinational conglomerate operating in the transport industry HNA group, which implemented a blockchain-enabled e-commerce platform with three main uses: issuing cryptocurrency, protecting sensitive business information, and lowering the boundaries between different business units [15].

Even though blockchain is considered a robust and secure technology, it still has certain vulnerabilities. The authors of [16] examine these vulnerabilities and find that the crucial risks are related to the *51% attack*, where the attacker controls the majority of the nodes in the network, the double spending attack, or the problem of insufficient private key security. Moreover, other more advances issues are due to vulnerable or under-optimised smart contracts, which result to stolen or wasted funds. The paper further considers attacks independent from the use of blockchain, such as selfish mining attack, where the attacker does not immediately broadcast a new valid block that was mined, leading to unnecessary work performed by the other nodes in the network, or the Eclipse attack, which consists of preventing a particular network node from receiving up-to-

date messages from the network and therefore limiting its functionality and/or performance.

In 2014, a security weakness was exploited in a popular cryptocurrency exchange, namely Mt. Gox, which led to subsequent crash of this exchange [2]. While this case was widely covered by the mainstream media, there were numerous other cases that didn't make the news, where the exchange was attacked and users lost their funds. The authors of [17] provide a comprehensive overview over these attacks, and conclude that the probability of a security breach in a cryptocurrency exchange is positively correlated with the transaction volume of that exchange.

All the cryptocurrency exchanges mentioned in [17] were "traditional", in a way that they require the user to deposit funds to the exchange and withdraw the exchanged funds afterwards. To the best of our knowledge, there is no fully decentralised implementation of cryptocurrency exchange available today. There are many exchange platforms offering trading Ethereum based tokens[4], but these tokens do not exist outside Ethereum realm. The only existing proposal for a trust-less digital assets exchange is introduced by Hallgren et al. [18], but this proposed architecture does not completely remove the system centralisation issue, since funds still need to be deposited to the Hallex system. Conversely, our solution is a fully decentralised cryptocurrency exchange platform, built on blockchains and smart contract technology.

The problem of interaction of two separate blockchains is examined closely in [19], where the author introduces several ways to enable cross-chain operability. These include notaries (a trusted group of entities, which operate and exchange assets on both blockchains) and relays (where one of the blockchains carries out a task to learn the status of the other blockchain). A detailed proposal of a notary has been described in [20]. A relay has been described and implemented in the BTC Relay system[5], which allows users to pay for execution of smart contracts on the Ethereum network with Bitcoins. While the exchange of the funds on different blockchains is not the primary aim of this system, it could be extended to support this use case.

3 Design and Implementation

To avoid the necessity of trust between the two trading parties, we propose an architecture where a smart contract functions as a digital escrow holding service. An escrow is a piece of property, temporarily held by a third party until a certain condition has been met[6]. The smart contract therefore holds user's funds, until a condition agreed upon by both users is true. An example of such a condition could be: *"Ether funds can only be released to Bob, if Bob has transferred his Bitcoin to Alice"*. In this scenario, after an initial agreement between Alice and Bob,

---

[4] https://angel.co/0xproject/, https://idex.market/

[5] http://btcrelay.org/

[6] https://www.investopedia.com/terms/e/escrow.asp

Alice proceeds to deploy the smart contract with an embedded condition to the network. She then transfers her Ether to this smart contract, where it will pose as an escrow. This transaction, together with the smart contract code is public and can be verified by Bob. When Bob sees that the funds are deposited in the smart contract, he can transfer his Bitcoins to Alice in an ordinary transaction. The smart contract queries the Bitcoin blockchain for Bob's transaction, and once it has been completed, the smart contract releases Alice's funds to Bob. If Bob tries to cheat Alice by not sending Bitcoins, the condition specified in the smart contract would not be met and the smart contract would not release the funds to Bob. If Alice attempts to cheat Bob by deploying a smart contract with incorrect condition, Bob would discover this by exploring the Ethereum blockchain. A trading scenario between Alice and Bob is illustrated in Figure 2, while in Figure 3 we provide a high-level overview of our system architecture. The logic of the smart contract is described in Figure 1.

```
class my-smart-contract {
        receive funds;
        check transaction status;
        if (transaction happened):
                forward funds to destination;
        else
                return funds to sender;
}
```

Fig. 1. High-level notation of the logic inside a half black-box.

The complexity of implementing this scenario lies mainly in step number 4, where the smart contract must verify the Bitcoin transaction. The difficulty arises from the fact that Ethereum and Bitcoin blockchains are two separate entities that operate on different networks, with different protocols, and are isolated in their respective realms. Indeed, the smart contract resides on the Ethereum blockchain, and to verify Bob's transaction it would need to get data from the Bitcoin blockchain. To address this problem, we make use of *oracles*, which are specialised data sources that translate real-world, off-chain information, into data that can be processed by the smart contract. Oracles watch the blockchain for specified events and respond by publishing the results of a query back to the smart contract [21]. A simple oracle could be a server that listens for these events on the Ethereum network, fetches the real-world data from the web or other location and then delivers the results back to the contract, sending them as a part of a transaction [22, 23].
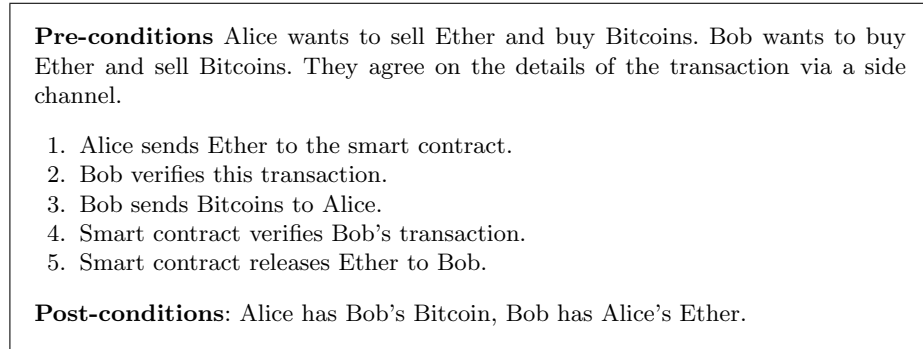

Pre-conditions Alice wants to sell Ether and buy Bitcoins. Bob wants to buy Ether and sell Bitcoins. They agree on the details of the transaction via a side channel.

1. Alice sends Ether to the smart contract.
2. Bob verifies this transaction.
3. Bob sends Bitcoins to Alice.
4. Smart contract verifies Bob's transaction.
5. Smart contract releases Ether to Bob.

Post-conditions: Alice has Bob's Bitcoin, Bob has Alice's Ether.

Fig. 2. Trading scenario where Alice and Bob trade Ethereum and Bitcoin with the help of the smart contract.

1. Initiate transaction
2. Transfer currency (BTC)
3. Query blockchain for payment
4. Return payment status
5a. Payment made: release funds to Actor B
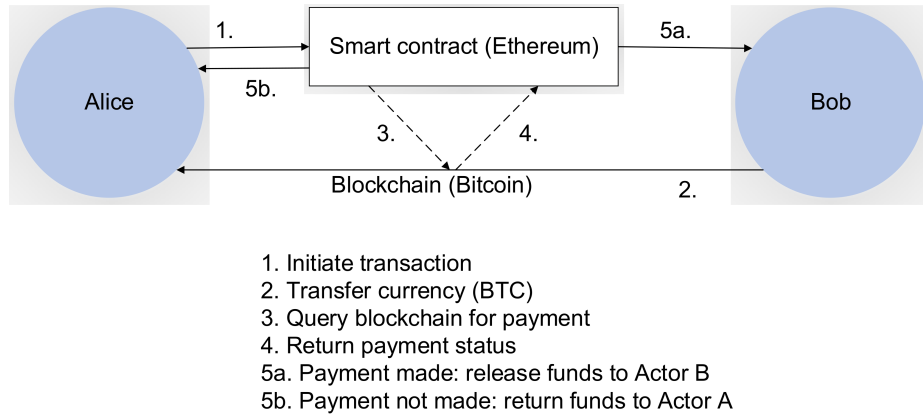5b. Payment not made: return funds to Actor A

Fig. 3. System architecture, reflecting the scenario in Figure 2.

Currently, there are several open source projects that provide *oracle* services – the biggest one being *Oraclize*[7], which offers simple integration with the Solidity language, which is one of the programming languages for smart contracts. Oraclize offers queries to multiple off-chain data sources, including simple HTTP queries or Wolfram Alpha. In our implementation we use Oraclize, however our platform is quite flexible, and other oracles can be integrated very easily.

3.1 Implementation of the System Components

The proposed system consists of several components that interact together, as presented in Figure 4. The main point of user interaction for both primary and secondary users is the system front-end, which communicates with the Ethereum blockchain via an Ethereum node to deploy a custom smart contract that oper-

---

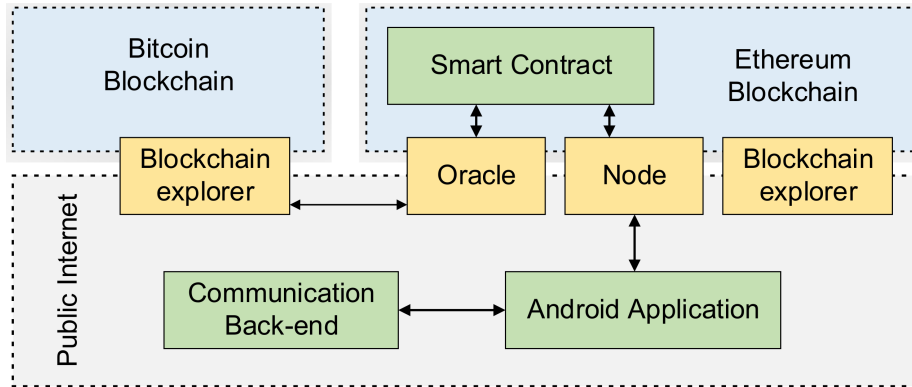[7] `http://www.oraclize.it` (`https://github.com/oraclize`)

Fig. 4. Overview of the system parts: *Android application* fetches the data about existing offers from the communication back-end and sends new smart contracts to the Ethereum node. *Node* communicates with other nodes in the Ethereum network, maintains the status of the blockchain and deploys new smart contracts to the network. *Smart contract* contacts the oracle after the deployment and holds the funds until the oracle has cleared the transaction as approved. The *oracle* queries the Bitcoin block explorer to learn about the status of the transaction and communicates the result back to the smart contract. *Communication back end* communicates only with the Android application. It holds details about users' offers and supports the trading interaction between the users.

ates the logic of the transaction, and with a back-end, which acts as a communication channel between users. To operate its logic, the smart contract communicates with an oracle, which queries a blockchain explorer provider to learn about the status of a transaction and sends the updates back to the smart contract.

Smart Contract. We program the contract using the Solidity language on Remix, an in-browser IDE, which allows sandbox contract testing[8]. Smart contracts written in Solidity get compiled into `BIN` (binary) and `ABI` (Application Binary Interface) format, using the *solc* compiler. The `BIN` file contains the Solidity code compiled into Ethereum Virtual Machine bytecode and the `ABI` file contains information about how to interact with the Smart Contract. The encoding of the `ABI` file is part of the Ethereum protocol specification. Then, we use the Web3j command line tools[9] to transform the `BIN` and `ABI` files into Java representation of the smart contract. The Java file includes wrapper methods to invoke the custom constructor and the methods of the contract.

Node. To be able to deploy the smart contracts, we need to have a node in the Ethereum blockchain network. For this purpose, we make use of Infuria, a node-hosting service[10]. Infura endpoints accept connections from any client and no authentication is needed. This is not an issue, since by design, our system

---

[8] `https://remix.ethereum.org/`

[9] `https://github.com/web3j/web3j`

[10] `https://infura.io/`

signs the transaction on device, before it is deployed to the network, meaning that no private information is uploaded on the node.

Oracle. We use the Oraclize service, as it is currently the most widely used in the community and includes extensive documentation and testing environment. The smart contract implements a `callback` method to register for replies from Oraclize, which is called by an Oraclize-issued smart contact together with the result. It is in this method that the smart contract handles the response.

Blockchain Explorer. This entity is needed for the oracle to understand if a transaction has been performed in the Bitcoin network. In our system, we use the *Simple Query API* provided by the Blockchain.info platform, even though any other service could be used without affecting the functionality of our platform.

Application Front-end. To demonstrate the functionality of the prototype, we implement an Android application, which provides a user interface that allows users to engage in a transaction and to deploy smart contracts on the network. It also provides guidance through the transaction process, displaying prompts to the users when their intervention is needed. Lastly, it provides a link to the smart contract, once it has been deployed, so that users can easily locate and verify it on the blockchain. To deploy the smart contract from the Android application, we use a special Java class, which holds the binary representation of the compiled Solidity code and wrapper methods that could be used to communicate with an already deployed smart contract.

Communication Back-end. The data processed by the communication back-end consists of users' Bitcoin and Ether offers. Besides storing users' offers, the back-end also serves as the coordination channel for the users. When a user performs an action in the application that advances the progress of the transaction, a particular field in the database is updated. To implement the back-end platform, we use Firebase, which is a noSQL database storing data in JSON format.

4 Evaluation

In this experiment we demonstrate a transaction between two persons: a primary user Eve, who wants to sell Ether and buy Bitcoin, and a secondary user Mike, who wants to buy Ether and sell Bitcoin. Both users run the application on their phones. Eve has her own Ethereum wallet with some Ether she wishes to sell, while Mike has his own Bitcoin wallet he will use to make the transaction. Eve's and Mike's wallets are standalone systems, independent from our system. The scenario of the transaction is as follows:

1. Mike creates a Bitcoin offer in the Android application by inputting:
   – Amount in Bitcoin he wishes to sell.
   – Amount in Ether he wishes to receive.
   – Ethereum address, where he wishes to receive the Ether.

Mike then confirms and the offer is pushed on the Firebase database.

2. Eve's device reacts to the change in the database and fetches the latest offers. Eve can now see Mike's offer. She selects it and confirms.

3. Eve is presented with a temporary Ethereum address that will be used to deploy the smart contract. To continue, Eve must transfer her funds to the temporary address. This address is generated on the device by our system, and no other party besides Eve's Android application has access to the private key.

4. Eve then creates an empty Bitcoin address where she will receive funds from Mike. She then inputs her public key to the Android application, so that Mike knows where to send the Bitcoin funds. Eve does not reveal her private key in the process.

5. A smart contract is deployed from the temporary Ethereum address with the following properties:
   – Mike's Ethereum address.
   – Eve's Bitcoin address.
   – Amount of Bitcoin, that needs to be received by Eve, before the smart contract releases the Ether.

   Besides these properties, which are specified in the smart contract constructor as data fields, the creation of a contract also carries two other significant information:
   – Sender's Ethereum address – this is used to return the funds in case of unsuccessful transaction.
   – Value, associated with the transaction – this is funds Eve is sending to Mike.

6. After the smart contract has been deployed, the transaction hash is displayed to both Mike and Eve. Mike needs this transaction hash to find and verify that Eve deployed the smart contract with the agreed properties.

7. After Mike verifies the correctness of the information in the smart contract, he proceeds transferring funds to the Bitcoin address in the smart contract.

8. Right after the deployment, the smart contract sends a query to Oraclize. The query consists of the following data:
   – Blockchain.info API endpoint reference.
   – Eve's Bitcoin address.
   – Delay, after which the API query is executed (600 seconds).

   Oraclize registers this query and starts a timer. After 10 minutes, it contacts the specified API endpoint with the specified Bitcoin address. This endpoint returns the unspent transaction output of the queried address in Satoshi as a plain string. After Oraclize receives this reply from the Blockchain.info, it passes the result to the smart contract by calling its `callback` method.

9. The contract compares whether the result fulfils the condition (i.e. if the unspent balance of the Bitcoin address received from the Blockchain.info is equal or greater than the balance specified by Eve, when she deployed the contract). If this condition is true, the smart contract sends its Ether balance to Mike. If it is false, it returns the Ether back to Eve.

This scenario was successfully executed with two Android devices, a Parity Ethereum wallet, and a GreenAddress Bitcoin wallet. The transactions were made on the Ethereum *Kovan* test network and Bitcoin *testnet3* test networks. Eve's device was a OnePlus A5000 running Android 8.1.0 Oreo, while Mike's device was a Samsung AM-A510F, running Android 7.0.0 Nougat and both devices had a stable internet connection. The whole process took approximately 20 minutes, out of which 10 minutes is the oracle response delay. The cost of the smart contract deployment was approximately 0.05 kETH[11] and was paid by Eve. This process can be replicated on the test networks at any time. The prerequisites are a balance of kETH and testnet BTC and transaction amount greater than 0.1 kETH.

5 Discussion

In this section, we present some challenges that need to be considered, which might prove to be crucial for the adoption of the system.

Cost of a Transaction. As mentioned earlier, there is a cost associated with every smart contract. The deployment of the smart contract used in the prototype uses approximately 2 million gas (as a comparison, a simple transfer of Ether uses exactly $21,000$ gas). The price payed for this amount of gas depends on the market conditions and on how fast the user wants the transaction to be processed. Limiting the amount of work preformed on the EVM by the smart contract can reduce the cost of the contract deployment and therefore the costs of the transaction. Moreover, it is also important to note that the smart contract we have used in this implementation is related to the API provided by Oraclize. It is possible that using another oracle provider might reduce the price for the smart contract deployment.

Contract Verification. The transaction process relies on the fact that the secondary user needs to verify the smart contract deployed by the primary user. While the smart contract can be found on the blockchain using the blockchain explorer, it is only the compiled bytecode that is being stored. As such, verifying that the contract is correct is not an easy task in the current version of our system, which needs to be simplified and automatised.

One Contract per Transaction. The current prototype always deploys a new contract whenever a transaction is made. This contract is then executed and discarded afterwards. Another approach would be to only deploy a single contract for the whole system. New transactions would simply call methods defined by this contract and the contract would be handling multiple transactions simultaneously. Naturally, the smart contract would need to be more complex, but the costs for its operation would be shared among all of its users.

Scalability Trading pair Bitcoin/Ethereum can be commonly found in the portfolio of digital currency exchanges. According to Coinmarketcap platform[12], the

---

[11] *kETH* is used to indicate Kovan Ether.
[12] https://coinmarketcap.com/

volume of the transactions between Bitcoin and Ether makes up to around 9% of the overall transaction volume across the 10 biggest digital currency exchanges. In the future, this number is likely to increase further, as cryptocurrencies further establish in the society. While this is not a problem for Ethereum and Bitcoin blockchains, it could affect our proposed system. With increasing number of queries, oracle(s) could change their pricing model which would further increase the transaction costs.

To handle larger number of users, the communication back-end would also need to be reconsidered, to allow interaction of many individuals, while not unnecessarily limiting the transaction making process.

6 Conclusions and Future Work

In this project, we proposed, designed, and implemented a system that enables a distributed and secure trade of cryptocurrencies between two parties. We built the system on the Ethereum platform and used smart contracts for the core of its operation. The smart contract holds the users' funds, until the other side of the transaction from the other party has been made. To demonstrate the process, we implemented an Android prototype, and showed that it is possible to successfully exchange Ethereum to Bitcoin, and vice-versa, without the need of a centralized service. Even though we performed the tests on the testing blockchain networks *Kovan* and *testnet3*, moving to the real ones is straightforward and is just a matter of deployment.

As plans of future development, we intend improving the system mainly on the following two aspects:

Use of Multiple Nodes. To avoid the single-node issue, since the node could refuse to perform a transaction or the provider could simply go offline, we will extend the system to use multiple nodes.

Use of Multiple Oracles and Multiple Blockchain Explorers. The current version of the system only uses one oracle and one blockchain explorer provider. Similarly as the node, they could prevent transactions from happening, if either of the two goes offline. A possible solution is to use multiple oracles, which in turn query different blockchain explorers.

References

1. Chaum, D.: Blind Signatures for Untraceable Payments. In Chaum, D., Rivest, R.L., Sherman, A.T., eds.: Advances in Cryptology, Boston, MA, Springer US (1983) 199–203
2. Popper, N., Abrams, R.: Apparent Theft at Mt. Gox Shakes Bitcoin World - The New York Times (2 2014)
3. McIntosh, R.: How to Choose Crypto Exchanges, Store Money and Avoid Scams. Finance Magnates (1 2018)

4. Chen, L., Xu, L., Gao, Z., Shah, N., Lu, Y., Shi, W.: Smart Contract Execution - the (+-)-Biased Ballot Problem. In Yoshio Okamoto; Takeshi Tokuyama, ed.: 28th International Symposium on Algorithms and Computation (ISAAC 2017). Volume 92., Dagstuhl, Germany, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017) 21:1– 21:12
5. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System
6. Robert Hackett: J.P. Morgan Chase Is Building an Ethereum-Based Blockchain: Here's Why (2016)
7. Swan, M.: Blockchain : blueprint for a new economy. 1 edn. O'Reilly Media, Incorporated (2015)
8. Michael, N., Gomber, P., Oliver, H., Dirk, S.: Blockchain. Business & Information Systems Engineering **59**(3) (6 2017) 183–187
9. Buterin, V.: A next-generation smart contract and decentralized application platform (2014)
10. Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.: Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. In Clark J., Meiklejohn S., R.P.W.D.B.M.R.K., ed.: Financial Cryptography and Data Security. Springer, Berlin, Heidelberg (2 2016) 79–94
11. Mik, E.: Smart contracts: terminology, technical limitations and real world complexity. Law, Innovation and Technology **9**(2) (2017) 269–300
12. Patrick, M., Shahandashti, S.F., Feng, H.: A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In Kiayias, A., ed.: Financial Cryptography and Data Security, Cham, Springer International Publishing (2017) 357–375
13. Dannen, C.: Introducing Ethereum and Solidity. Apress, Berkeley, CA (2017)
14. Mansfield-Devine, S.: Beyond Bitcoin: using blockchain technology to provide assurance in the commercial world. Computer Fraud & Security **2017**(5) (5 2017) 14–18
15. Ying, W., Jia, S., Du, W.: Digital enablement of blockchain: Evidence from HNA group. International Journal of Information Management **39** (4 2018) 1–4
16. Li, X., Jiang, P., Chen, T., Luo, X., Wen, Q.: A survey on the security of blockchain systems. Future Generation Computer Systems (8 2017)
17. Moore, T., Christin, N.: Beware the Middleman: Empirical Analysis of Bitcoin-Exchange Risk. In Sadeghi, A.R., ed.: Financial Cryptography and Data Security, Berlin, Heidelberg, Springer Berlin Heidelberg (2013) 25–33
18. Hallgren, J., Hallgren, M., Fisher, S., Larsen, N., Hautop, J., Ross, O.: Hallex: A Trust-Less Exchange System for Digital Assets. SSRN Electronic Journal (2 2017)
19. Buterin, V.: Chain interoperability (2016)
20. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments. draft version 0.5 **9** (2016) 14
21. John Weldon: Building an "Oracle" for an Ethereum contract (2016)
22. : Oraclize documentation (2018)
23. Jules Dourlens: Oracles: bringing data to the blockchain (2017)