Application and Evaluation of Payment Channel in Hybrid Decentralized Ethereum Token Exchange

by

Xuan Luo

B.E., Tongji University, China 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

October 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, a thesis entitled:

Application and Evaluation of Payment Channel in Hybrid Decentralized Ethereum Token Exchange

submitted by Xuan Luo in partial fulfillment of the requirements for

the degree of Master of Applied Science

in Electrical and Computer Engineering

Examining Committee:

Victor C.M. Leung, Electrical and Computer Engineering

Supervisor

Zehua Wang, Electrical and Computer Engineering

Supervisory Committee Member

Jane Z. Wang, Electrical and Computer Engineering

Additional Examiner

Abstract

As the killer decentralized application hosted by blockchain, cryptocurrencies have been accepted as digital cash by many investors and consumers nowadays. Token is a special kind of cryptocurrencies hosted on a blockchain. Due to the popularity of Ethereum, many token exchange platforms are for Ethereum based tokens nowadays. In general, we classify current token exchange supporting exchange of Ethereum based tokens into three categories, i.e., centralized token exchange (CEX), decentralized token exchange (DEX), and hybrid decentralized token exchange (HEX).

CEX has been suffering from hacking due to the centralized management of users' tokens. In contrast, DEX maintains users' assets by smart contracts in a decentralized manner, but introduces additional overhead in terms of monetary and waiting time (*i.e.*, *gas fee* and *transaction confirmation latency*). HEX has been proposed to combine the benefits of CEX and DEX. However, existing HEX is criticized for two issues. First, trading transactions are time-consuming and expensive for frequent token traders. Second, excessive simultaneous transactions might cause the transaction congestion in Ethereum.

In this thesis, we propose a payment channel based HEX, which extends existing solutions by adding a new payment channel layer to benefit frequent traders and alleviate the potential transaction congestion. We represent the system architecture of the proposed HEX and compare it with the conventional HEX to show how payment channel helps in decreasing the number of on-chain transactions for frequent traders.

Besides, we propose the very first gas-price vs. transaction-confirmation-latency function to guide Ethereum transaction issuers to determine an optimal gas price that minimizes the overall cost. Based on the proposed gas-price vs. transaction-confirmation-latency function, we quantitatively evaluate the performance of payment channel in HEX by comparisons of the cost in terms of gas fee and transaction confirmation latency for a user in the conventional HEX and the proposed payment channel based HEX when the user's overall cost is minimized. Simulation results demonstrate the effectiveness of our proposed mechanism in terms of reducing gas fee and transaction confirmation latency for frequent traders as well as the potential transaction congestion in Ethereum.

Lay Summary

We classify current token exchange supporting exchange of Ethereum based tokens into three categories, i.e., centralized token exchange (CEX), decentralized token exchange (DEX), and hybrid decentralized token exchange (HEX). CEX is criticized for its security and privacy issues. DEX solves these issues by introducing additional trading cost in terms of monetary and waiting time to the system. HEX has been proposed to combine the advantages of CEX and DEX. However, existing HEX is unfriendly for frequent traders and may cause the transaction congestion in Ethereum.

In this thesis, we aim to reduce the cost in terms of monetary and waiting time for frequent traders as well as the potential transaction congestion in Ethereum by adding a new payment channel layer in HEX. Also, we quantitatively evaluate the performance of payment channel in HEX. Experimental results demonstrate the effectiveness of the payment channel based HEX.

Preface

This thesis is an original and independent work by the author, Xuan Luo. The research problem, problem formulation, and experiment design came from my own efforts with the help of other collaborators. Chapters 2–3 encompass the works that have been published or currently under review. The corresponding papers are under the supervision of Professor Victor C.M. Leung and Professor Zehua Wang and the collaboration with Professor Wei Cai and Professor Xiuhua Li. The collaborator's contributions are as follows:

1. Professor Victor C.M. Leung provided valuable guidance in identifying the research problems and developing solution methodologies.

2. Professor Zehua Wang provided constructive advice on the research problem in Chapter 2 and helped me formulate the optimization problem in Chapter 3.

3. Professor Wei Cai and Professor Xiuhua Li provided helpful comments for improving the paper related to Chapters 2 and 3.

For all chapters, I hereby declare that I am the first author of the corresponding papers. These papers are listed as follows:

Journal Papers, Submitted

- Xuan Luo, Zehua Wang, Wei Cai, Xiuhua Li, and Victor C.M. Leung, "Application and Evaluation of Payment Channel in Hybrid Decentralized Ethereum Token Exchange," submitted, 2019.

Conference Papers, Published

- Xuan Luo, Wei Cai, Zehua Wang, Xiuhua Li, and Victor C.M. Leung, "Application of payment channel in hybrid decentralized Ethereum token exchange," in *Proceedings of IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, Seoul, South Korea, May 2019.

Table of Contents

Abstract . . . . . **iii**

Lay Summary . . . . . **v**

Preface . . . . . **vi**

Table of Contents . . . . . **viii**

List of Tables . . . . . **xi**

List of Figures . . . . . **xii**

List of Abbreviations . . . . . **xiv**

Acknowledgements . . . . . **xv**

Dedication . . . . . **xvi**

1 Introduction **1**

1.1 Introduction of Ethereum Token Exchange 1

1.2 Related Work 6

1.2.1 State-of-the-art Token Exchange 6

1.2.2 Payment Channel 8

1.3 Motivations and Contributions 10

1.4 Thesis Organization 11

2 System Overview 12

2.1 Introduction 12

2.2 System Architecture 12

2.2.1 On-chain Layer 13

2.2.2 Payment Channel Layer 16

2.2.3 Off-chain Layer 17

2.2.4 Comparisons with Existing Solutions 18

2.3 Security Analysis 22

2.3.1 Security Assumptions 22

2.3.2 Attack Vectors 22

2.4 System Implementation 25

2.4.1 On-chain layer 25

2.4.2 Off-chain layer 28

2.5 Summary 29

3 Performance Evaluation 30

3.1 Introduction 30

3.2 Minimizing overall cost for frequent traders 30

3.2.1 Overall Cost Modelling 31

3.2.2 Parameter Estimation 35

3.2.3 Simulation Results 43

3.3 Experiment 46

3.3.1 Experimental Setup 46

3.3.2 Experimental Cases 49

3.4 Summary 55

4 Conclusions and Future Work 57

Bibliography 59

Appendices 63

Appendix A: Smart contract for the payment channel based HEX 63

Appendix B: Smart contract for the conventional HEX 75

List of Tables

2.1 Comparison of order placement 21
2.2 Comparison of order cancellation 21
2.3 Comparison of order matching 22

3.1 Model parameters and mixing probabilities 41
3.2 List of hardware settings 46
3.3 List of software settings 47
3.4 Gas costs in payment channel creation and closure 49

List of Figures

2.1 Architecture of the payment channel based HEX 13

2.2 An illustration of payment channel between Alice and the HEX 17

2.3 An example for how a trader exchanges tokens with the HEX without payment channel. For continuous trades, a user needs to repeat Steps 2, 5, and 6. 19

2.4 An example for how a trader exchanges tokens with the payment channel based HEX. For continuous trades, a user needs to repeat Steps 7, 8, 9, 10, and 11. 19

2.5 Illustration of software utilization 25

2.6 The user interface of the payment channel enabled HEX 26

3.1 Regression models for clusters using finite mixture of linear regressions. Models are listed in descending order of the intercept b. 42

3.2 Gas-price vs. transaction-confirmation-latency model 44

3.3 Overall cost versus gas price when users change the maximum gas fee and transaction confirmation latency for a given transaction. Here we take gas cost = 21000, $\alpha = 0.5$ as an example. 45

3.4 Overall cost versus gas price when users' preference between gas fee and transaction confirmation latency changes for a given transaction. Here we take gas cost = 21000, maximum gas fee = 0.0282 Ether, and maximum transaction confirmation latency = 710 sec as an example. 46
3.5 Gas fee saving ratio versus number of trading transactions per payment channel. 50
3.6 Latency saving ratio versus number of trading transactions per payment channel. 52
3.7 Blockchain transaction saving ratio versus average number of trading transactions per payment channel. 54

List of Abbreviations

API	Application programming interface
AWS	Amazon Web Services
BIC	Bayesian Information Criteria
CEX	Centralized token exchange
CLTV	CheckLockTimeVerify
DEX	Decentralized token exchange
EM	Expectation-maximization
HEX	Hybrid decentralized token exchange
KYC	Know-Your-Customer
MVP	Minimal Viable Program
PoW	Proof of Work

Acknowledgements

First and foremost, I would like to express my gratitude to my parents for their unconditional love and support. Without their continuous support and encouragement, I could not have completed this thesis.

Second, I would like to thank my supervisor, Professor Victor C.M. Leung, for his patience, encouragement, and invaluable advice during my master study. I thank him for providing me with an excellent research atmosphere in his lab. I also greatly thank my collaborator Professor Zehua Wang for his constructive advice and comments for my research.

Also, I would like to thank the member of my master committee, Professor Jane Z. Wang, for the time and effort in evaluating my work and providing feedback and suggestions.

Many thanks go to all my colleagues and friends.

Dedication

To my families

Chapter 1

Introduction

1.1 Introduction of Ethereum Token Exchange

A blockchain [1] is a continuously growing list of linked blocks shared by all nodes across a peer-to-peer network, where each block typically contains a cryptographic hash of the previous block, a timestamp and transaction data. A transaction is usually a record of an exchange of digital currency between two users. When new transactions are submitted to a blockchain, these transactions will be broadcast to all nodes in the peer-to-peer network. Those nodes called miners collect new transactions and work on generating a new block which meets the rules of the consensus algorithm of the blockchain based on the collected transactions. In blockchain, the consensus algorithm allows the majority of all nodes to reach consensus on the generation and validation of a new block. Take the widely used consensus algorithm Proof of Work (PoW) [2] as an example. In a PoW based blockchain, all miners compete to be the first to solve a hard cryptographic puzzle by brute-forcing and produce a winning value. The process of solving the cryptographic puzzle is known as mining. When a miner finds a winning value and generates a new block based on the winning value, it broadcasts the block to all nodes. Other miners confirm the validity of the received new block according to the rules of the consensus algorithm, and accepts the new block by generating the next block in the chain using the hash of the accepted block

as the previous block's hash. The block that contains new transactions is now added to the blockchain, and these transactions inside the new block are considered as receiving one block confirmation. Each new block added afterwards is another confirmation of the transaction.

However, due to the concurrent nature of the mining process, more than one miner might be able to find a winning value at the same time. Each miner that finds a winning value may broadcast its proposed new block to the blockchain, which may lead to different branches of linked blocks. Some miners add blocks to one branch, while other miners add blocks to other branches, based on which miner with winning value is closest to them. The consensus algorithm ensures that only the longest branch will get included in the blockchain and an eventual consistency could be achieved among all nodes regarding the state of the blockchain. But in such cases, a transaction that only receives one block confirmation is subject to reversal by the blockchain network. A malicious user might submit a modified copy of the transaction to the blockchain right after the original transaction has been broadcast in the blockchain. If the modified copy is included into the longest branch and the original transaction is included into a discarded branch, the original transaction is discarded and reversed. As the difficulty to reverse a transaction increases exponentially with the number of block confirmations, a blockchain transaction issuer usually waits for a certain number of block confirmations to make sure his/her transaction is unlikely to be reversed. Thus, blockchain transaction issuers suffer from *transaction confirmation latency*. Transaction confirmation latency is the interval between the submission time when a transaction is submitted to blockchain and the confirmation time when the transaction receives a certain number of block confirmations in blockchain. For different blockchains, the number of block confirmations needed for blockchain transaction issuers are different. The larger the number of block confirmations is, the longer transaction confirmation latency

is.

Blockchain allows the transactions between users to be carried out in a decentralized manner without third-party's intervention. However, the decentralization nature of blockchain also brings some new issues. First, a blockchain transaction issuer needs to pay for a *transaction fee* when submitting a transaction to blockchain. The transaction fee is an incentive for the miners to include the transaction into their blocks. Second, a blockchain transaction issuer needs to wait for a certain amount of time to make sure his/her transaction is confirmed and unlikely to be reversed in blockchain. Block generation costs some time since miners need some time to solve the cryptographic puzzle in PoW. To make sure the transaction is unlikely to be reversed, the blockchain transaction issuer needs to wait for the generation of a certain number of blocks. Third, due to the fact that the number of transactions could be included into a block is limited, the transaction throughput is relatively low in blockchain. Thus, excessive simultaneous transactions might cause the transaction congestion in blockchain.

There are mainly two generations of the blockchain. The first generation of blockchains, introduced by Bitcoin [3], serve as public ledgers for monetary transactions. The second generation of blockchains enable not only the record of monetary transactions, but also the carrying out of computational results. Compared to the first generation of blockchains, the second generation of blockchains have wider functionality. For example, the most popular second generation blockchain — Ethereum [4], could serve as a public ledger that records both monetary transactions and computational results. In Ethereum, transaction fee is called *gas fee* that a blockchain transaction issuer paid in a special resource called *gas*, which is the "fuel" for computational instructions executed in Ethereum. Gas fee could be estimated by multiplying *gas price* and *gas cost*. Gas price is the unit cost of gas and gas cost is the amount of gas needed to execute a transaction. As for transaction confirmation

latency, since usually Ethereum transaction issuers wait for 12 block confirmations, transaction confirmation latency in Ethereum is the interval between the submission time when a transaction is submitted to Ethereum and the confirmation time when the transaction receives 12 block confirmations.

As the killer decentralized application hosted by blockchain, cryptocurrencies [5] have been accepted as digital cash by many investors and consumers nowadays. Token is a special kind of cryptocurrencies which represents a particular fungible and tradable asset or a utility hosted on a blockchain. Every token belongs to a blockchain address and is only accessible by the person who has the private key related to the blockchain address. Tokens could be classified based on the blockchain that host them, for instance, Bitcoin based tokens and Ethereum based tokens.

According to CoinMarketCap [6], more than 90% of the top 100 cryptocurrencies are Ethereum based tokens. Thus, many token exchange platforms where people could conveniently exchange tokens with others are for Ethereum based tokens nowadays. In this thesis, our focus is on the studies of token exchange for Ethereum based tokens. Generally, we classify current token exchange supporting the exchange of Ethereum based tokens into three categories, *i.e.*, centralized token exchange (CEX), decentralized token exchange (DEX), and hybrid decentralized token exchange (HEX).

CEX is the most popular token exchange ever since its presence. The core idea of CEX is centralization, which is achieved in the way that users are required to deposit tokens to a CEX-provided blockchain address and thus all users' assets are taken care of by the CEX instead of by users themselves. The centralization nature of the CEX brings two benefits to users. First, CEX provides a rapid trading speed since any trading transaction leads to a direct update on the CEX database instead of an update on the blockchain. Second, CEX gives users the option to trade in private, since trading records of a particular user

will not be shown publicly in blockchain. However, the centralization nature of CEX is contradictory with the decentralization spirit of the blockchain and distributed ledger. It also makes CEX intrinsically vulnerable to hacking and denial of service attacks.

Different from centralized management of users' tokens in the CEX, DEX lets users manage their own cryptocurrencies and acts as a transparent middle-man for users. In the DEX, trading procedures are implemented in smart contracts [7], which are scripts recorded on the blockchain and executed automatically without third party's intervention. Users conduct their trades by addressing a blockchain transaction to trigger a procedure defined in the smart contract. DEX minimizes the potential security issue in the CEX by leveraging automatically executed smart contracts, but it also introduces three critical issues. First, the lack of order matching service makes it difficult for token traders to find appropriate counterparties. Second, all trading requests are now blockchain transactions, which imply more cost in terms of monetary and waiting time, *i.e.*, transaction fee and transaction confirmation latency. Third, due to the low transaction throughput in Ethereum caused by PoW, excessive simultaneous trading transactions might cause the transaction congestion in Ethereum.

HEX is a hybrid approach that combines the benefits of CEX and DEX. HEX addresses the trade discovery issue by maintaining a centralized order management database, while all trades are still conducted by calling procedures in the smart contracts. However, this approach does not fix the second issue of high cost in terms of gas fee and transaction confirmation latency introduced by those frequent on-chain transactions. This is particularly important for frequent token traders, as more blockchain transactions imply more gas fee and longer transaction confirmation latency. Also, this approach does not solve the potential transaction congestion problem caused by excessive simultaneous transactions.

In this thesis, we propose to extend existing HEX solutions by adding a new payment

channel [8] layer to decrease the total gas fee and transaction confirmation latency for frequent traders as well as alleviate the potential transaction congestion caused by excessive simultaneous trading transactions. The payment channel is a technique allowing for off-chain payments with a final on-chain settlement [9]. By utilizing payment channel technique, only two on-chain transactions will be posted to the blockchain, while the token traders could make (nearly) unlimited number of off-chain payments. Thus, the proposed payment channel based HEX eliminates the number of on-chain transactions for frequent traders, which decreases the total gas fee and transaction confirmation latency brought by on-chain trading transactions for frequent traders, and alleviates the potential transaction congestion. In this chapter, we introduce the fundamentals related to this thesis, including the state-of-the-art token exchange and the overview of payment channel.

The rest of this chapter is organized as follows. In Section 1.2, a literature survey on the existing research of token exchange and the payment channel technique is presented. Motivations and major contributions of the thesis are concluded in Section 1.3. The organization of the thesis is given at the end of this chapter.

1.2 Related Work

1.2.1 State-of-the-art Token Exchange

Mt.Gox [10] is the first famous CEX being introduced to the public since the prevalence of Bitcoin. Unfortunately, there are not just theoretical risks but disasters that have occurred for thousands of cryptocurrency investors in the past. In 2014, Mt.Gox claimed that 850,000 bitcoins belonging to customers and the company were missing, which led to the loss of thousands of customers. However, CEX is still the mainstream of token exchange nowadays. Popular token exchange, including Coinbase [11], Gemini [12], Poloniex [13], Kraken [14], and Huobi [15], still adopts the CEX architecture.

Relying on smart contracts, DEX provides more secure trading services to the cryptocurrency users. DEX like KyberNetwork [16] and AirSwap [17] decentralizes the settlement and all related functions by defining them as the on-chain procedures. However, DEX is still unpopular among investors, due to the lack of centralized order management. Since DEX uses on-chain orderbook to manage orders, DEX users have to monitor the on-chain orderbook from time to time in order to find potential matching orders. Besides, since all trading transactions are now blockchain transactions, this increases the cost in terms of gas fee and transaction confirmation latency for token traders, and might cause the potential transaction congestion in Ethereum.

Combining the benefits of CEX and DEX, HEX represents the latest version of token exchange. EtherDelta [18] and 0x Project [19] are described as decentralized token exchange but they are more like a hybrid design. They decentralize the settlement and use a centralized server to handle the orderbook. However, as one successful trade requires three blockchain transactions, users of EtherDelta and 0x Project will pay for extra gas fee and have to wait for confirmations of three blockchain transactions. Besides, there is no order matching engine provided, so the buyers have to monitor the market all the time. Also, many buyers may compete for one token sell order, and the unsuccessful buyers waste their gas fees. IDEX [20], JOYSO [21] and DEx.top [22] improve the above issues with an automatic order matching engine. Especially, JOYSO claims that traders can continue trading on JOYSO without the successful confirmation of the previous trading transaction on the blockchain. Yet, this causes a potential security issue, where a failure of the broadcast of a matching order to blockchain will invalidate all dependent orders. Moreover, hacking of the token exchange may cause the loss of all users' assets.

However, none of the above can provide friendly user experience for frequent traders.

1.2.2 Payment Channel

A payment channel is the technique designed to allow users to make a series of off-chain payments. So, it can facilitate multiple cryptocurrency trades without committing all trade records to blockchain one by one. With a typical payment channel, only two on-chain transactions will be posted to the blockchain, while (nearly) unlimited number of off-chain payments between the participants can happen in the middle of the two on-chain transactions. For instance, Alice creates a channel to Bob by initiating an on-chain deposit transaction to deposit tokens into the deployed smart contract that serves as an escrow account. Alice can then make an arbitrary number of rapid payments to Bob, by creating and sending signatures of the latest deposit division agreements to Bob over the Internet. The signatures sent from Alice and received by Bob are not tokens, but Bob can eventually receive the tokens by closing the payment channel. When Bob closes the payment channel, the latest deposit division signature created by Alice is required to invoke the procedure in the smart contract to distribute the deposit to counterparties' accounts. This is the reason that we call the deposit division signatures sent from Alice to Bob the off-chain payments. Note that the smart contract will not receive any off-chain payments. Hence, these intermediate off-chain payments will not be processed by the blockchain miners.

Payment channels can be classified into two types, namely, the uni-directional and bi-directional payment channels. A uni-directional payment channel only allows single directional off-chain payment. In CLTV-style uni-directional payment channel [23], the security of the payment channel is based on the fact that the payment receiver does not have the incentive of using an old signature issued by the spender to close the payment channel. On the other hand, a bi-directional payment channel allows both counterparties to send off-chain payments. Therefore, one may have the incentive of using an old signature issued by the counterparty to close the payment channel and receive more asset than he/she deserves.

Thus, new security models are required to resolve this problem. The duplex payment channel [24] uses time-locking transactions to resolve the above problem to an extent. The newest signature is always created with a shortest time-lock, meaning that it is the first one that triggers the channel closing procedure. While, due to the intrinsic mechanism, duplex payment channels have limited lifetime. Poon-Dryja payment channel in the Lightning Network [8] solves the lifetime issue in the duplex payment channel by taking advantage of multi-signature and hashed time-lock contract technologies. The security of Poon-Dryja payment channel is based on the punishment of a malevolent counterparty rather than on time. A challenging-period will be specified by both counterparties when creating the payment channel. When a dishonest counterparty tries to close the payment channel by submitting its opponent's payment signature to blockchain, if the opponent could submit a newer payment signature received from the dishonest counterparty within the challenging-period, the dishonest side would be punished and all deposits in the escrow account could be transferred to the honest side.

Payment channels have been widely used in off-chain peer to peer micropayments, like in the Lightning Network. But such application of payment channel in the Lightning Network is not suitable in HEX due to the following two reasons: First, most of the time a token trader will not trade with the same opponent continuously. If a trader needs to establish a new payment channel when there is no existing payment channel to the target trader, the overhead of creating and closing the payment channel could be not economical. Second, Lightning Network only supports exchanging two types of tokens within the entire network, while in practical token exchange, it is common for users to exchange more than two types of tokens within a certain period. Therefore, in order to decrease the overhead of our proposed scheme, users will establish a payment channel with the token exchange instead of with other users directly. Such application of payment channel in HEX also

allows for multi-token exchange.

1.3 Motivations and Contributions

In this thesis, we extend existing HEX solutions by adding a new payment channel layer to reduce gas fee and transaction confirmation latency for frequent token traders and the potential transaction congestion in Ethereum. We represent the system design of the proposed solution and also validate the effectiveness of the payment channel based HEX by the experimental comparisons with conventional HEX in this thesis. The major contributions of the work are as follows:

- We systematically design the very first payment channel based HEX to benefit frequent traders. The proposed scheme supports multi-token off-chain exchange, and decreases the total gas fee and transaction confirmation latency for frequent traders. Also, the proposed scheme could ease the potential transaction congestion caused by excessive simultaneous trading transactions in Ethereum.

- We propose the very first gas-price vs. transaction-confirmation-latency function to help blockchain transaction issuers to decide an optimal gas price that minimizes the overall (*i.e.*, monetary and waiting time) cost.

- Based on the proposed gas-price vs. transaction-confirmation-latency function, we quantitatively evaluate the performance of the payment channel based HEX. The gas-price vs. transaction-confirmation-latency function can also be used to quantitatively evaluate the performance of payment channel in general applications using the payment channel technique.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. In Chapter 2, we firstly present the system architecture of the proposed payment channel based HEX system. We then compare the proposed payment channel HEX with existing solutions in terms of system working mechanism and order execution model. Afterward, we analyze the security of the system in terms of the security assumptions and potential attacks towards the HEX and the HEX users. In the end, we introduce a prototype implementation of the proposed payment channel based HEX. In Chapter 3, we introduce the very first gas-price vs. transaction-confirmation-latency function first. Then, we show how the overall cost could be minimized based on the proposed gas-price vs. transaction-confirmation-latency function. Afterward, we quantitatively evaluate the performance of payment channel in HEX by comparing the cost (*i.e.*, gas fee and transaction confirmation latency) for users in the conventional HEX and the proposed HEX when the overall cost is minimized. Finally, the thesis is concluded and some potential future work is introduced in Chapter 4.

Chapter 2

System Overview

2.1 Introduction

In Chapter 1, we have introduced to add a new payment channel layer to the HEX to benefit frequent traders and alleviate the potential transaction congestion in Ethereum. In this chapter, we present the system overview of the proposed payment channel based HEX.

The rest of this chapter is organized as follows. In Section 2.2, we present the architecture of the proposed payment channel based HEX system, followed by the comparisons between the conventional HEX and the proposed payment channel based HEX. Security analysis of the proposed solution is provided in Section 2.3. The implementation of a Minimum Viable Program (MVP) could be found in 2.4. The chapter is summarized in Section 2.5.

2.2 System Architecture

Figure 2.1 illustrates the proposed system framework, which consists of three layers, namely, the on-chain, payment channel, and the off-chain layers. The on-chain layer works as a trustworthy escrow account to secure assets of the HEX and the HEX users. The payment channel layer is the bridge to connect the on-chain layer with the off-chain layer.

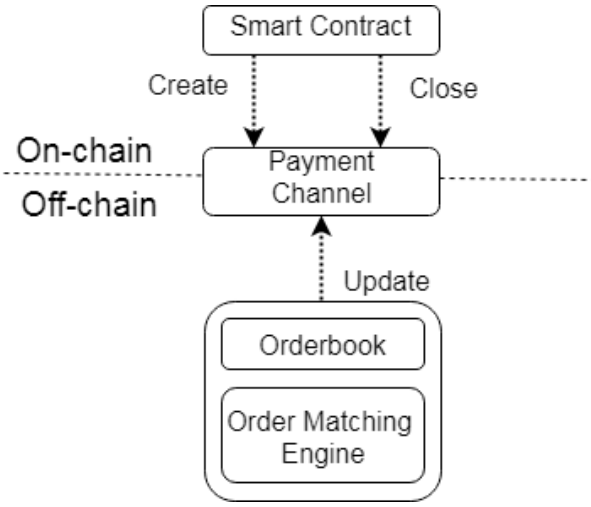


Figure 2.1: Architecture of the payment channel based HEX

The off-chain layer is responsible for order execution.

2.2.1 On-chain Layer

The on-chain layer is the key to securing users' assets as well as creating and closing a payment channel. The deployed smart contract in the on-chain layer acts as a verifiable, open source, and trustworthy escrow account.

Before a HEX user is able to exchange tokens in the HEX platform, both the user and the HEX deposit tokens into the smart contract first. Then, a payment channel creating transaction with both counterparties' signatures has to be submitted to the blockchain in order to create a payment channel. Afterward, the user and the HEX could exchange tokens off-chain by exchanging signatures to update the deposit division agreement in an off-chain manner. When a user wants to withdraw his/her deposit, the user could submit a signed payment channel closing transaction to trigger the payment channel closing procedure. Settlement can thus be done on-chain in the smart contract, which cannot be altered or interfered with.

To introduce the on-chain layer in details, the main functions of the smart contract are defined as follows:

- **Deposit**: Both the user and the HEX send their deposits to the smart contract. Since smart contract cannot be altered or interfered with, it acts like a trustworthy escrow account.

- **Withdraw**: A user is only allowed to withdraw his/her deposit that is not locked in the payment channel. If a user wants to withdraw the deposit that is locked in the payment channel, the user needs to close the payment channel in order to do so.

- **Create a payment channel**: To create a payment channel with the HEX, a user shall firstly sign a time-locked message to create a payment channel with a maximum lifetime and a challenging-period. After the time specified by the time lock, the message becomes invalid and could not be used to create a payment channel any more. The maximum lifetime of the payment channel determines the duration that the user could exchange tokens in the HEX platform after a payment channel has been created. The challenging-period specifies a time duration after a counterparty requests to close the payment channel, and in this duration, the opponent can dispute. The user sends the payment channel creating signature to the HEX, and the HEX exchanges its own signature with the user if the HEX agrees to create the payment channel with the user. The user then needs to send both counterparties' signatures to the blockchain to trigger the creation of a new payment channel in the smart contract before the time specified by time lock.

- **Top up a payment channel**: When a user wants to add some more deposit for exchange while holding a payment channel with the HEX, the user could deposit into the smart contract and lock the new deposit into the existing payment channel. Also, the HEX can top up an existing payment channel by locking some more deposit into the payment channel.

- **Close a payment channel**: There are three scenarios where a payment channel can be closed:
  - Both the user and the HEX agree to close a payment channel. In this scenario, no matter if the payment channel reaches its maximum lifetime or not, both counterparties exchange their signatures of the agreement on payment channel closure. Then, the user or the HEX can send the two-party signed payment channel closing message to the smart contract and the locked deposits in the payment channel are split and saved in both counterparties' accounts on-chain at once.
  - Only one counterparty wants to close the payment channel before the maximum lifetime of the payment channel. In this case, the payment channel can also be closed but not so straightforward as above. Without loss of generality, we assume Alice is an HEX user and wants to close the payment channel with the HEX. Alice can use the off-chain payment signature received from the HEX to *request* to close the payment channel unilaterally by calling a procedure in the smart contract. The smart contract first verifies the signature and then set the status of the payment channel to "requested-to-close". The payment channel enters the challenging-period. Within the challenging-period, if the HEX could send the smart contract a signature issued by Alice and newer than that signature used by Alice to request to close the channel, Alice is proved cheating. The predefined penalty subroutine could be triggered and the payment channel can be closed with bias. Otherwise, if no dispute happens, the payment channel can be closed after the challenging-period. The assets in the payment channel are split according to what revealed by the signature that Alice used to request to close the payment channel.

– Only one counterparty wants to close the payment channel after the maximum lifetime of the payment channel. After the payment channel has reached its maximum lifetime, one counterparty could trigger the close of the payment channel with its own signature. Then, the deposits in the locked payment channel will be distributed according to the initial deposit division. Such design is to minimize users' loss when potential attacks towards users occur. This will be introduced later with more details when analyzing the security of the proposed HEX.

2.2.2 Payment Channel Layer

Payment channel layer is the bridge to connect the on-chain layer with the off-chain layer. Since creating a payment channel requires deposits from both counterparties, the main usage of the payment channel is to establish the trustworthiness between the counterparties, so that they can conduct off-chain trades by exchanging the signatures of the deposit division.

Here, we choose the bi-directional payment channel instead of the uni-directional payment channel as the user and the HEX need to exchange tokens with each other. For example, Alice wants to exchange tokens in the HEX for three times, the detailed process of which is shown in Figure 2.2. Both Alice and the HEX need to deposit into the smart contract and sign an initial deposit division agreement in order to create a payment channel. The initial deposit division agreement is the Trade 0 in the Figure 2.2. Then, Alice exchanges tokens three times. In Trade 1, Alice exchanges 0.5 ETH for 5 VERI. In Trade 2, Alice exchanges 5 VERI for 50 OMG. In Trade 3, Alice exchanges 10 OMG for 100 REP. Then, the payment channel is closed by sending a two-party signed payment channel closing transaction based on the latest deposit division agreement to the blockchain.

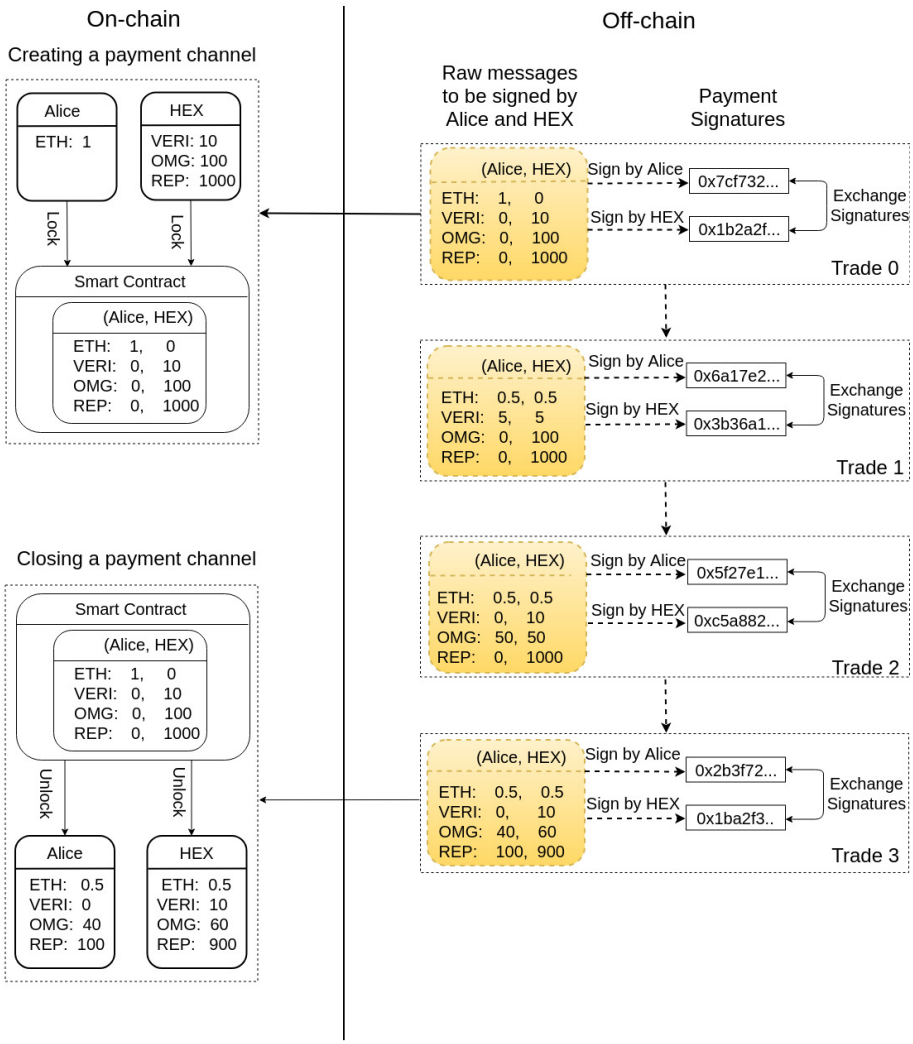


Figure 2.2: An illustration of payment channel between Alice and the HEX

2.2.3 Off-chain Layer

The off-chain layer is responsible for order execution, which includes order placement, order cancellation, and order matching.

After a payment channel is created between the user and the HEX, the user is able to continuously trade with the HEX off-chain. To start a new trade, the user could place an order by sending a signed buy/sell order to the HEX. The signature of the buy/sell order here is only used to show the user's willingness to trade and not used to update the deposit

division agreement in the on-chain escrow account. The new order will then be recorded in the orderbook. When the HEX finds a matching order via order matching engine, the HEX would then notify the user immediately and ask the user to reply within a specific time. Afterward, the user needs to send a payment signature to the HEX before the time specified by the HEX. The HEX then exchanges its payment signature to complete the order. If the user does not reply within the time specified by the HEX, the related order will be cancelled by the HEX. If a user wants to proactively cancel the order, the user needs to sign and send another order-cancel request to the HEX before the HEX finds a matching order for the user's order.

2.2.4 Comparisons with Existing Solutions

To help readers better understand our proposed HEX system, we compare our system with existing solutions in two aspects, *i.e.*, working mechanism of the system and order execution model.

Working Mechanism

To compare the working mechanism, Figure 2.3 and Figure 2.4 illustrate the key components and continuous trading workflows of the conventional HEX and the proposed payment channel based HEX, respectively. According to Figure 2.3 and Figure 2.4, it is obvious that the proposed HEX adds a payment channel layer to alter the workflow of trading procedures.

As shown in Figure 2.3, a user in the conventional HEX deposits tokens into the smart contract before starting a trade. Then, the user sends a signed token buy/sell order to the HEX. If the HEX finds a matching order, the HEX signs both the buy and sell orders issued by two token traders to approve the trade. Afterward, the signed matching orders will be submitted to the blockchain by the HEX for deposit settlement. For continuous

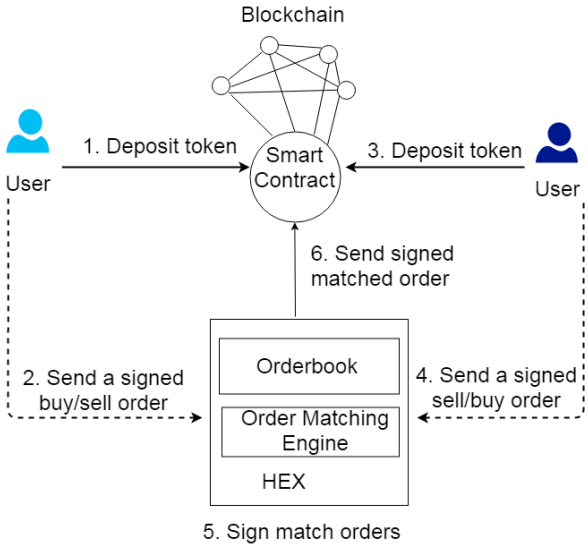


Figure 2.3: An example for how a trader exchanges tokens with the HEX without payment channel. For continuous trades, a user needs to repeat Steps 2, 5, and 6.

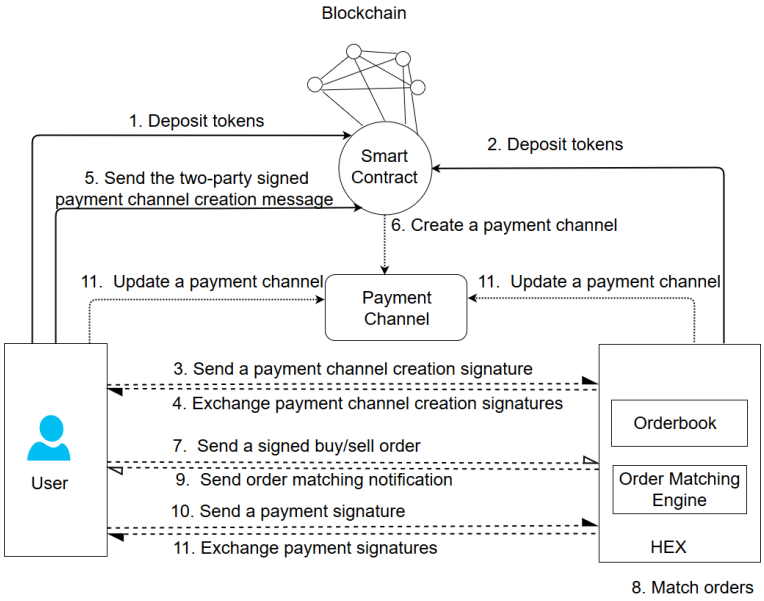


Figure 2.4: An example for how a trader exchanges tokens with the payment channel based HEX. For continuous trades, a user needs to repeat Steps 7, 8, 9, 10, and 11.

trades, a user needs to repeat Steps 2, 5, and 6 in Figure 2.3.

As shown in Figure 2.4, a user in our proposed system also deposits tokens into the deployed smart contract first. While, before the user can conduct the off-chain token exchange, he/she needs to sign and exchange a payment channel creating message with

the HEX in order to create a payment channel. After a payment channel is created, the user could place signed buy/sell orders to show his/her willingness to exchange tokens. If a matching order could be found by the HEX, both the seller and the buyer then exchange with the HEX their payment signatures that reveal their agreements on new deposit divisions for their deposits in the on-chain escrow account. For continuous trades, a user needs to repeat order placement and payment signature exchange in Steps 7, 8, 9, 10, and 11 in Figure 2.4.

According to the workflows of continuous trades, Step 6 in Figure 2.3 is on-chain, while Steps 7, 8, 9, 10, and 11 in Figure 2.4 are completely off-chain. Thus, the proposed payment channel based HEX eliminates the number of on-chain transactions for continuous trades, and achieves better performance over conventional HEX in terms of total gas fee and transaction confirmation latency.

Order Execution Model

To compare the execution model, we look at the following aspects: whether order placement is on-chain and whether order cancellation and matching are supported.

- **Order placement**: Orders in token exchange could be classified into two types in terms of the impact on cryptocurrency liquidity, *i.e.*, the make order and the take order. In the context of token exchange, cryptocurrency liquidity refers to the ability of a token to be converted into other tokens easily. The make order is a buy/sell order that will not be fulfilled immediately, and thus, adds liquidity to an exchange. While, the take order is a buy/sell order that will be immediately fulfilled, and thus, decreases liquidity of an exchange. Table 2.1 shows whether make and take orders are placed on-chain in different token exchange.

- **Order cancellation**: As shown in Table 2.2, we compare how order cancellation

Table 2.1: Comparison of order placement

Exchange Type	**Make Order**	**Take Order**
CEX	Off-chain	Off-chain
DEX	On-chain	On-chain
Conventional HEX	Off-chain	Off-chain
Proposed HEX	Off-chain	Off-chain

takes place in different token exchange. In CEX, the conventional HEX, and the proposed HEX, order cancellation is free since orders are placed off-chain and could be cancelled off-chain. While, users in DEX have to send an order cancelling transaction to the DEX smart contract to cancel an order, and thus, it costs extra Ethereum gas fee. To successfully cancel an order, users in CEX and the conventional HEX need to rely on the token exchange, while users in DEX and the proposed HEX could cancel the order at their will before the order is fulfilled.

Table 2.2: Comparison of order cancellation

Exchange Type	**Cost**	**Trust Model**
CEX	Free	Trustful
DEX	Gas fee	Trustless
Conventional HEX	Free	Trustful
Proposed HEX	Free	Trustless

- **Order matching**: Table 2.3 shows whether order matching is supported in different token exchange. DEX does not support order matching as it utilizes a decentralized orderbook on the blockchain to record orders. While, order matching is supported in CEX, the conventional HEX, and the proposed payment channel based HEX, since all of them record orders using centralized orderbooks.

Table 2.3: Comparison of order matching

Exchange Type	Order Matching Support
CEX	Yes
DEX	No
Conventional HEX	Yes
Proposed HEX	Yes

2.3 Security Analysis

In this section, we introduce the security assumptions and potential attacks in the proposed payment channel based HEX.

2.3.1 Security Assumptions

We have the following assumptions for the proposed payment channel based HEX system:

- The HEX is always online to provide service to the HEX users.
- The user needs to be notified when the HEX requests to close the payment channel. Techniques like WatchTower [25] could help here to alert the user when the HEX tries to close the payment channel unilaterally.
- The user or the HEX is able to close a payment channel before the maximum lifetime of the payment channel.

2.3.2 Attack Vectors

Here, we classify attacks into two types, *i.e.*, attacks towards users and attacks towards the HEX.

Attacks towards users

- **Signature Holding Attack**: During the order fulfilling process, the HEX could hold the payment signature from a user. When the HEX sends the order matching

notification to the user, the user will send a payment signature based on the trading agreement to the HEX. The HEX could keep the payment signature and become unresponsive to the user. In such situation, the user would immediately know that the HEX might be malicious. Since the user is able to cancel all other orders without agreement from the HEX, the user could simply wait till the maximum lifetime of the payment channel. This will lead to following two possible results:

  - The HEX might use the latest payment signature to close the payment channel before the maximum lifetime of the payment channel. In this situation, the user could get his/her deposit back based on the latest deposit division agreement.
  - The HEX might not close the payment channel proactively. In this situation, the user could trigger the close of the payment channel after the payment channel reaches its maximum lifetime. All trades during the lifetime of the payment channel become invalid since the payment channel settles the deposit based on the initial deposit division as the payment channel is created.

  In both situations, the user could either get his/her deposit based on the latest deposit division agreement or get his/her initial deposit back. But for the HEX, it will lose the trust of the user, which in long term decreases liquidity of the exchange. Thus, for the HEX side, they will not have enough incentive to hold the payment signatures issued by the users to cheat.

- **Front-running Attack**: Front-running is an investing strategy that predicts the impact of upcoming trades using prior trading information about their own or others. Front-running attack [26] might be launched by the HEX to the users since the HEX could get private knowledge of an upcoming order and strategically place their own orders in such a way that is profitable for the HEX. Even though the HEX could

theoretically front-run users by itself, users could detect that orders were not in sequence. The loss of the trust from users would result in bad liquidity of the HEX in long term, which makes it fair to say that the HEX is unlikely to front-run users for a long term purpose.

Attacks towards HEX

- **Private Key Hacking Attack**: The HEX might lose part of its deposit held by the smart contract if its private keys were stolen. Deposit held by the smart contract could be classified into two types, *i.e.*, deposit locked in the payment channel and deposit not locked in the payment channel. Since the smart contract could not be altered, deposit in the smart contract could only be withdrawn by using private keys. If an attacker steal the private keys of the HEX, he/she could transfer the deposit of the HEX which is not locked in the payment channel into his/her private address at once. However, deposit locked in the payment channel could only be withdrawn till the close of the payment channel. To close a payment channel, the attacker also needs to hack the off-chain HEX platform to get the latest payment signatures issued by the HEX user. Even assuming that the attacker is able to hack both the private keys of the HEX and payment signatures from HEX users, the close of the payment channel is an on-chain operation, which requests confirmations from the blockchain to take effect and this will cost some time. During this time, the token exchange could have some emergency solutions. For example, when the HEX is hacked, the system administrator of the HEX would be able to update the address of the HEX to a new address for all existing payment channels, thus, all HEX's deposits in these payment channels will be sent to this new address when payment channels are closed. It is worth noting that users' deposits are safe no matter if the HEX is hacked or not. An attacker will not be able to steal users' deposits in the smart contract as long as

he/she does not hack the private keys of users.

- **Liquidity Attack**: Malicious users may lock a large amount of tokens from the HEX without placing any order or placing orders that are hard to be fulfilled. As the locked tokens of the HEX could not be used to serve other users who actively exchange tokens, the liquidity of the HEX will decrease in long term. This could be solved by setting KYC (Know-Your-Customer) [27] or an effective incentive-and-punishment mechanism.

2.4 System Implementation

We implement a MVP for the proposed payment channel based HEX. We first introduce the enabling technologies, and then present the implementation details of the MVP. The related code could be found in the Appendix A.

As shown in Figure 2.5, the system can be implemented in two parts: the implementation of the smart contract for the on-chain layer, and the implementation of contract function calls for the off-chain layer.

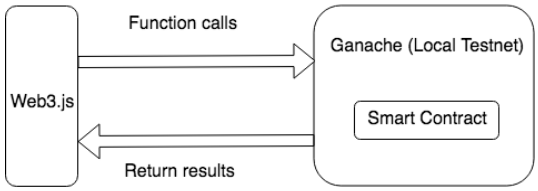


Figure 2.5: Illustration of software utilization

2.4.1 On-chain layer

For the on-chain layer, we adopt the Solidity [28] programming language to implement the Ethereum smart contract. To set up the testing environment, we deploy the proposed token exchange to Ganache, a local Ethereum blockchain powered by Truffle framework [29].

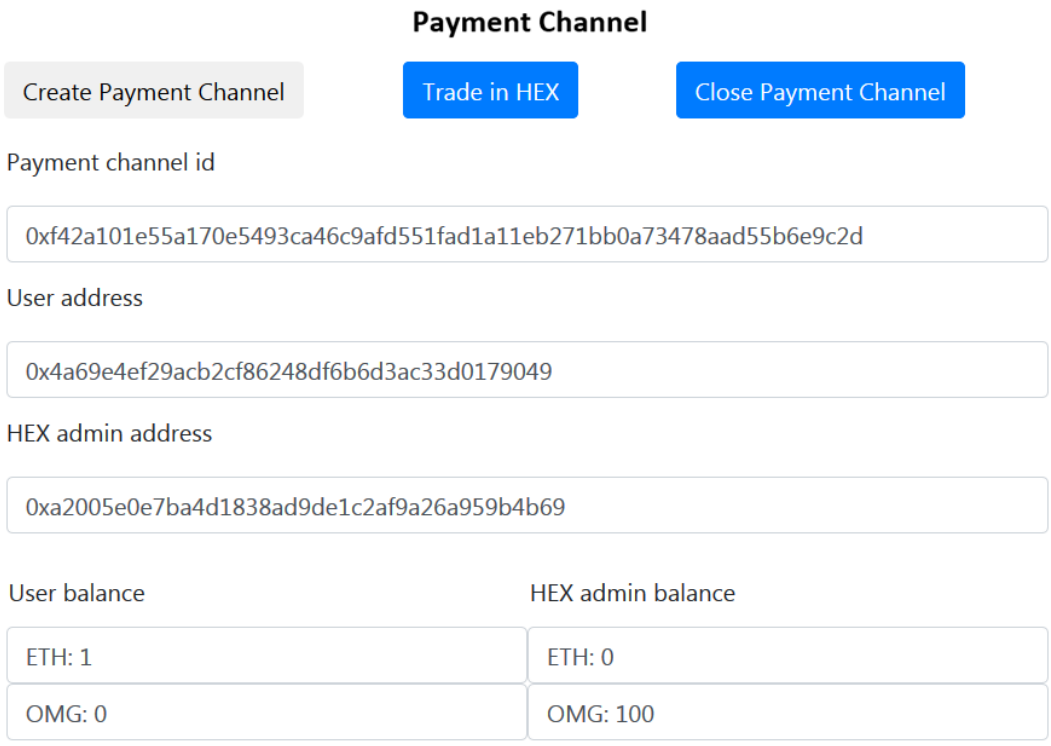


Figure 2.6: The user interface of the payment channel enabled HEX

Followings are the most two important functions in the smart contract of token exchange: payment channel creation and closure.

Payment Channel Creation

For the creation of a payment channel, a payment channel creating transaction is signed by both the user and the HEX, and is submitted to trigger the payment channel creating procedure defined in the smart contract.

As shown in Algorithm 2.1, if both the user and the HEX have enough deposits in the smart contract, a payment channel could be created by locking both deposits of the user and the HEX into the payment channel.

As shown in Figure 2.6, after a payment channel has been successfully created, the information of the new payment channel like the initial deposit division agreement will be updated in the HEX system so that the user could further trade with the HEX.

Algorithm 2.1 Create a payment channel.

Input: A two-party signed payment channel creating transaction.
Check the validity of the signature of $user$.
if not a valid signature **then**
    $revert()$.
end if
Check the validity of the signature of $admin$.
if not a valid signature **then**
    $revert()$.
end if
for every supported token **do**
    if $userBalance > availableTokens[targetToken][user]$ **then**
        $revert()$.
    else if $userBalance < 0$ **then**
        $revert()$.
    else if $adminBalance > availableTokens[targetToken][admin]$ **then**
        $revert()$.
    else if $adminBalance < 0$ **then**
        $revert()$.
    else
        Lock the fund of the token exchange for the payment channel.
    end if
end for
$channels[id] \leftarrow channel$.
$channelUsers[user] \leftarrow id$.
Emit event $CreatePC(id, user)$.

Payment Channel Closure

For the closure of a payment channel, we take the situation when both the user and the HEX agree to close the payment channel as an example.

As shown in Algorithm 2.2, when both the user and the HEX agree to close the payment channel, a payment channel closing transaction is signed by both counterparties and is submitted to trigger the payment channel closing procedure defined in the smart contract. After successful process, the payment channel will be closed and deposits in the payment channel are redistributed to both counterparties according to the deposit division agreement in the payment channel closing transaction.

Algorithm 2.2 Close a payment channel when both parties agree.

Input: A two-party signed payment channel close transaction.
Check the validity of the signature of *user*.
if not a valid signature **then**
    revert().
end if
Check the validity of the signature of *admin*.
if not a valid signature **then**
    revert().
end if
for every supported token **do**
    if $userBalance < 0$ **then**
        revert().
    else if $adminBalance < 0$ **then**
        revert().
    else if $userBalance + adminBalance \neq totalBalance$ in payment channel **then**
        revert().
    else
        Reset *userBalance* and *adminBalance* according to the signed transaction.
    end if
end for
delete $channels[id]$.
delete $channelUsers[user]$.
$numChannels - -$.
Emit event $ClosePC(id, user)$.

2.4.2 Off-chain layer

For the off-chain layer, we skip the detailed implementation of order placement and order matching functions in the proposed HEX, as they are well-developed in conventional HEXs. Instead, we adopt web3.js [30], the official Ethereum invocation JavaScript API for front-end developers, as the Ethereum client to trigger the function calls from off-chain layer to smart contract. We take token deposit function as an example to illustrate the usage of web3.js.

As shown in Algorithm 2.3, a payment channel instance of the proposed HEX is deployed firstly. After that, the HEX admin and the user deposit tokens to the smart contract by calling the function *depositToken*.

Algorithm 2.3 Deposit Token Javascript.

1: Deploy an instance of payment channel based HEX.
2: Deploy instances of supported tokens.
3: *admin* calls smart contract method *depositToken*.
4: Wait for the 12 block confirmations of the deposit transaction.
5: *user* calls smart contract method *depositToken*.
6: Wait for the 12 block confirmations of the deposit transaction.

2.5 Summary

In this chapter, we proposed to add a new payment channel layer in the HEX which eliminates the number of on-chain transactions for continuous traders, and thus, benefits frequent token traders in terms of gas fee and transaction confirmation latency as well as alleviates the potential transaction congestion in Ethereum. We firstly presented the system architecture of the proposed payment channel based HEX system. We further compared the conventional HEX with the proposed payment channel based HEX in terms of working mechanism and order execution model, and thus, showed how the number of on-chain transactions for continuous trades could be decreased in the proposed HEX. Afterward, we presented the security analysis of the proposed payment channel based HEX and showed the detailed implementation of a MVP for the proposed payment channel based HEX.

Chapter 3

Performance Evaluation

3.1 Introduction

In Chapter 2, we have introduced the system design of the proposed payment channel based HEX. In this chapter, we quantitatively evaluate the performance of payment channel in HEX by comparing the cost (*i.e.*, gas fee and transaction confirmation latency) for users in the conventional HEX and the proposed HEX when the overall cost is minimized.

The rest of this chapter is organized as follows. In Section 3.2, we introduce the very first gas-price vs. transaction-confirmation-latency function, and we show how the overall cost could be minimized based on gas-price vs. transaction-confirmation-latency function. Simulation results are provided in Section 3.3. The chapter is summarized in Section 3.4.

3.2 Minimizing overall cost for frequent traders

In this section, we propose the very first gas-price vs. transaction-confirmation-latency function first. Based on the function, we use the weighted sum of gas fee and transaction confirmation latency to model the overall cost for a HEX user, and formulate the overall cost minimization problem as a convex optimization problem. Then, we introduce finite mixture of linear regressions to qualitatively estimate the unknown parameters in the proposed gas-price vs. transaction-confirmation-latency function. Afterward, we run simulations to show

how a HEX user's overall cost changes with the gas price when the user changes his/her preference.

3.2.1 Overall Cost Modelling

There are two major cost for blockchain transaction issuers, *i.e.*, gas fee and transaction confirmation latency, which could be evaluated by the following performance metrics, respectively:

- Gas fee: to describe the cost (unit: Ether) in a blockchain transaction.
- Transaction confirmation latency: to describe the interval (unit: sec) between the submission time when a transaction is submitted to blockchain network, and the confirmation time when the transaction receives 12 block confirmations in blockchain.

For the gas fee, the gas to be used by a blockchain transaction could be regarded as a constant since it could be estimated by calling Ethereum API like web3.eth.estimateGas [30]. Therefore, for a transaction, the gas fee could be expressed by

$$f_1(x) = c \times x/10^9 \tag{3.1}$$

where

- $f_1(x)$ denotes gas fee;
- c denotes gas cost, which describes the amount of gas to be used by a blockchain transaction;
- x denotes gas price, which describes the price per unit of gas with the unit 1×10^9 Wei (*i.e.*, GWei).

For the transaction confirmation latency, it is well known that as with the increase of gas price, the transaction confirmation latency will decrease. Based on this, we propose the following gas-price vs. transaction-confirmation-latency function for Ethereum Mainnet

$$f_2(x) = a/x + b \tag{3.2}$$

where

- $f_2(x)$ denotes the estimation of the transaction confirmation latency;
- a denotes the competitive pressure how soon transactions could be mined in Ethereum Mainnet. When a is small, it means that the competitive pressure is low and thus the change of gas price will not cause big change in transaction confirmation latency. While, when a is very big, it means that the competitive pressure is high, and thus, the change in gas price will lead to a considerable change in transaction confirmation latency;
- b denotes the estimation of transaction confirmation latency in an ideal situation where x is large enough so that a transaction will be mined in no time;
- x denotes gas price, which describes the price per unit of gas with the unit 1×10^9 Wei (*i.e.*, GWei).

It is worth mentioning that we will estimate the parameters (a, b) in the proposed function as well as validate the effectiveness of the proposed function via big data analysis later.

For a blockchain transaction issuer, the objective is to simultaneously minimize gas fee and transaction confirmation latency, which could be regarded as a multi-objective optimization problem [31]. The multi-objective optimization problem could be turned into a single-objective mathematical optimization problem which is to minimize the overall

cost of gas fee and transaction confirmation latency via weighted sum model [32]. As gas fee and transaction confirmation latency have different magnitude, normalization is required for gas fee and transaction confirmation latency when utilizing weighted sum model. According to [33], there are three major methods to normalize objective functions in weighted sum model, *i.e.*, normalize by the magnitude of the objective function at the initial point, normalize by the minimum of the objective function, and normalize by the differences of optimal function values in the Utopia and Nadir points, where Utopia and Nadir points define the lower and upper bounds of the optimal Pareto set for objective functions, respectively. In our case, it is hard to determine a proper initial point of both gas fee and transaction confirmation latency, and the minimum of the gas fee could be 0, which makes normalization by the initial point and by the minimum of the objective functions impractical and ineffective. Therefore, normalization by Utopia and Nadir points is chosen in our case.

Theoretically, the maximum gas fee and transaction confirmation latency could be huge. While, in real world, users cannot use infinite gas fee in a transaction or wait endlessly for the confirmation of a transaction. Thus, for a given user we could define the maximum gas fee as f_1^{max}, and the maximum transaction confirmation latency as f_2^{max}.

Combining (3.1) and (3.2), we could derive the following equations:

$$\begin{aligned} f_1^U &= \frac{a \times c}{10^9 \times (f_2^{max} - b)}, \\ f_1^N &= f_1^{max}, \\ f_2^U &= \frac{a \times c}{10^9 \times f_1^{max}} + b, \\ f_2^N &= f_2^{max} \end{aligned} \tag{3.3}$$

where

- f_1^U denotes the lower bound of $f_1(x)$ based on the Utopia point of $f_1(x)$;
- f_1^N denotes the upper bound of $f_1(x)$ based on the Nadir point of $f_1(x)$;

- f_2^U denotes the lower bound of $f_2(x)$ based on the Utopia point of $f_2(x)$;
- f_2^N denotes the upper bound of $f_2(x)$ based on the Nadir point of $f_2(x)$.

Thus, for a blockchain transaction issuer with the maximum gas fee as f_1^{max} and the maximum transaction confirmation latency as f_2^{max}, the minimization of the overall cost in terms of gas fee and transaction confirmation latency could be reformulated as

$$\min_x \left\{ \frac{\alpha \times c}{10^9 \times (f_1^{max} - f_1^U)} \times x + \frac{(1-\alpha) \times a}{(f_2^{max} - f_2^U)} \times \frac{1}{x} - \frac{\alpha \times f_1^U}{(f_1^{max} - f_1^U)} - \frac{(1-\alpha) \times (f_2^U - b)}{(f_2^{max} - f_2^U)} \right\},$$
$$\text{subject to: } x > 0 \tag{3.4}$$

where

- We use the lower bounds based on Utopia points in (3.3) to simplify the expression of the single objective function;
- α is a weighting coefficient, which stands for the blockchain issuer's preference of gas fee over transaction confirmation latency, and $0 < \alpha < 1$.

We denote the above single objective function as $f(x)$. To minimize $f(x)$, let us have a look at the first and second derivatives of $f(x)$ as follows:

$$\begin{aligned} f'(x) &= \frac{\alpha \times c}{10^9 \times (f_1^{max} - f_1^U)} - \frac{(1-\alpha) \times a}{(f_2^{max} - f_2^U)} \times \frac{1}{x^2}, \\ f''(x) &= \frac{2 \times (1-\alpha) \times a}{(f_2^{max} - f_2^U)} \times \frac{1}{x^3}. \end{aligned} \tag{3.5}$$

It is obvious that $f''(x) > 0$ always hold, thus, we could know that $f(x)$ is minimized when and only when x makes $f'(x) = 0$. By solving the equation $f'(x) = 0$, the optimal gas price to minimize the overall cost could be expressed as

$$x^* = \sqrt{\frac{10^9 \times (1-\alpha) \times a \times (f_1^{max} - f_1^U)}{\alpha \times c \times (f_2^{max} - f_2^U)}}. \tag{3.6}$$

However, to make use of the overall cost function, we still need to know the value of parameters (a, b) in gas-price vs. transaction-confirmation-latency function. In next section, we will use big data analysis technique to get qualitative estimates of of parameters (a, b).

3.2.2 Parameter Estimation

To estimate the parameters (a, b) in the gas-price vs. transaction-confirmation-latency function, we collect $540,296$ transactions from Ethereum Mainnet. Based on these transactions, we could get $540,296$ (x, y) pairs, where

- x denotes gas price, which describes the price per unit of gas with the unit 1×10^9 Wei (*i.e.*, GWei);
- y denotes transaction confirmation latency.

To calculate the transaction confirmation latency for a transaction, we need to know the submission time when a transaction is submitted to blockchain, and the confirmation time when a transaction is confirmed by 12 blockchain blocks. The confirmation time is the block time of the 12^{th} block that confirms the transaction, which could be easily obtained by calling Ethereum API [30]. As for the submission time, the simplest way to get it is to use an Ethereum client to submit transactions to Ethereum by ourselves, and thus, we could get the accurate submission time of these transactions. However, it cost real money to submit Ethereum transactions in Ethereum Mainnet. A large number of transactions means a large amount of real money needed for collecting the submission time of these transactions, and the cost is especially high for transactions with a high gas price.

Therefore, it is economically inaffordable to collect the submission time of transactions by submitting transactions to Ethereum MainNet by ourselves. In this thesis, we use the observation time when a transaction is firstly observed by an Ethereum full node to approximate the transaction's submission time. To be specific, we use the observation time when a transaction firstly shows up in the pending transaction pool of an Ethereum full node to approximate the submission time of the transaction. However, due to the unavoidable network latency in gossip protocols, the observation time of a transaction would deviate from the submission time depending on the network distance between the Ethereum full node who observes the transaction and the Ethereum client who submits the transaction. To minimize the deviation caused by the network latency, we make the following assumptions:

Assumptions

- We assume the network latency between two network servers using gossip protocols to exchange information could be classified into different network latency zones according to the value of the network latency.

- Assume the maximum number of network latency zones is K. Based on the network latency between the Ethereum full node who observes transactions and the Ethereum clients who submit these transactions, the collected transactions would fall in K different network latency zones. That is to say, the collected transactions will belong to one of clusters $1, 2, .., K$.

- We define that the lower cluster number is, the lower network latency is. Since transactions from cluster 1 have the minimal network latency, we will use the model from the cluster 1 to approximate our target model.

- For clusters $1, 2, 3, ..., K$, we assume the following

$$\begin{cases} y_1 = a_1/x_1 + b_1 + \epsilon_1, \\ ... \\ y_k = a_k/x_k + b_k + \epsilon_k, \\ ... \\ y_K = a_K/x_K + b_K + \epsilon_K \end{cases} \tag{3.7}$$

  where

  - x_k denotes gas price for a transaction in cluster k, which describes the price per unit of gas with the unit 1×10^9 Wei (*i.e.*, GWei); $x_k > 0$, $k = 1, 2, ..., K$;
  - y_k denotes transaction confirmation latency for a transaction in cluster k, $k = 1, 2, ..., K$;
  - $a_k > 0$, $k = 1, 2, ..., K$;
  - ϵ_k is random error, under the assumption of normality, where $\epsilon_k \sim \mathcal{N}(0, \sigma_k^2)$, where σ_k is the standard deviation of the Gaussion distribution of ϵ_k, $k = 1, 2, ..., K$.

  Based on what we have explained before, we have

$$b_1 > b_2 ... > b_K. \tag{3.8}$$

  This can be deducted as follows: for all clusters, when a blockchain issuer would like to pay for a very high gas price, a transaction will be immediately included into a block by minerss. In this case, transaction confirmation latency will depend on block time and network latency. Since block time will not change for transactions from different clusters, estimation of transaction confirmation latency will change according to network latency only. As with the increase of cluster number, network

latency between the Ethereum full node who observes transactions and the Ethereum clients who submit transactions increases. Therefore, when gas price gets very high, as with the increase of cluster number, estimation of transaction confirmation latency will decrease due to the increase of network latency. Thus, we have $b_1 > b_2 ... > b_K$.

Data Collection and Pre-Processing

Here we introduce how we collect and pre-process data.

- We set up five Ethereum full nodes through Amazon Web Services (AWS) [34] in the following locations: West America, Southeast America, Europe, Northeast Asia Pacific, Southeast Asia Pacific. In each Ethereum full node, we record the observation time of transactions when these transactions firstly show up in the pending transaction pool of the node.

- Data collection starts at 2019-04-12 00:00:00 UTC and ends at 2019-04-14 23:59:59 UTC. To preserve as many transactions with long transaction confirmation latency as possible, we only use transactions which are mined successfully between 2019-04-14 00:00:00 UTC and 2019-04-14 23:59:59 UTC in our model fitting process. In this way, we are able to keep these transactions whose transaction confirmation latency is longer than two days.

- Transactions collected by all five nodes are merged based on the transaction hash. For transactions with the same transaction hash, only the one with the earliest observation time is kept. There are two reasons for this:

  – By merging transactions collected from five different areas of the world, the potential bias caused by different network structure in different areas of the world could be minimized.

  - By using the earliest observation time of transactions, the weights of transactions from clusters with low network latency are increasing, which also potentially increases the weight of transactions from cluster 1.

- After above processing, we get $540,296$ (x, y) pairs from transactions mined successfully on 2019-04-14 UTC.

Data Modelling

So far, we have a mixed collection of (x, y) pairs from K clusters, and we need to estimate the parameters (a_1, b_1) from the cluster 1. As shown in (3.7), our models are finite mixtures of regressions. Let $x' = 1/x$. By replacing x with x', we could turn our models into linear regressions. Afterward, we could fit the data with finite mixture of linear regressions [35], which could estimate the distinct parameters of each regression model within finite mixture models.

Given a set of independent observations $y_1, y_2, ..., y_n$, corresponding to values $x'_1, x'_2, ..., x'_n$, then the problem in (3.7) could be reformulated as

$$y_i = \begin{cases} a_1 \times x'_i + b_1 + \epsilon_{i1}, & \text{with probability } \pi_1, \\ ... \\ a_k \times x'_i + b_k + \epsilon_{ik}, & \text{with probability } \pi_k, \\ ... \\ a_K \times x'_i + b_K + \epsilon_{iK}, & \text{with probability } \pi_K \end{cases} \tag{3.9}$$

where

- x'_i, y_i is the i^{th} observation of the collected $(1/x, y)$ pairs, $i = 1, 2, ..., n$;

- π_k is the mixing probability where

  - $0 < \pi_k < 1$, for all $k = 1, 2, ..., K$;

  - $\sum_{k=1}^{K} \pi_k = 1$;

- ϵ_{ik} is random error, under the assumption of normality, where $\epsilon_{ik} \sim \mathcal{N}(0,\, \sigma_k^2)$, where σ_k is the standard deviation of the Gaussion distribution of ϵ_{ik}, $i = 1, 2, ..., n$, $k = 1, 2, 3, ...K$.

The complete parameter set of the mixture model $\theta = (\pi_1, \pi_2, ..., \pi_K, a_1, a_2, ..., a_K, b_1, b_2, ..., b_K, \sigma_1, \sigma_2, ..., \sigma_K)$ could be estimated by maximizing the log-likelihood

$$L\left(\theta | x_1', .., x_n', y_1, .., y_n\right) = \sum_{i=1}^{n} \ln \left(\sum_{k=1}^{K} \pi_k \phi\big(y_i | a_k, b_k, \sigma_k, x_i'\big) \right) \tag{3.10}$$

where

$$\phi\left(y_i | a_k, b_k, \sigma_k, x_i'\right) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \times \exp\left(-\frac{\big(y_i - (a_k x_i' + b_k)\big)^2}{2\sigma_k^2} \right). \tag{3.11}$$

The standard tool for maximum likelihood estimation in finite mixture models is the EM algorithm [36]. The EM algorithm is an iterative method to find maximum likelihood estimates of parameters in a two-step process. First, the E-step computes the conditional expectation of the log-likelihood evaluated using the current estimates for the parameters. Second, the M-step maximizes the log-likelihood of the parameter estimates learned in the E-step.

Once we have learned the model parameters based on EM algorithm, we then need to decide an optimal cluster number. A consistent estimator of the cluster number K could be achieved based on Bayesian Information Criteria (BIC) [37], where the value of K is chosen at which the BIC value asymptotically converges.

After getting the optimal number of clusters and related model parameters, we could choose the model of cluster 1 from models for all clusters based on (3.8).

Results

For the model fitting process using EM algorithm, we use the finite mixture models fitting package mixtools [38]. For the cluster number, we try K from 2 to 10. According to [39], the solution of the EM algorithm depends on the initial value and the stopping criterion. Therefore, for each given K, we start the EM iteration with 10 different initial values and choose the stopping criterion as when the difference between two iterations is less than 1×10^{-6}. Afterward, we choose the one with the largest log-likelihood as the solution for a given K within the 10 iterations. Then, we compare the BIC of each solution for K from 2 to 10, the BIC value converges when $K = 8$ in our experiment. Therefore, we choose our optimal cluster number as 8, and the optimal model fitting results of 8 clusters are shown in Figure 3.1. In Figure 3.1, latency refers to the transaction confirmation latency. Besides, related model parameters and mixing probabilities are shown in Table 3.1, where numbers in the column Parameters (a, b) are rounded to two decimal points. It is worth noting that in Table 3.1, the models are listed in descending order of the intercept b.

Table 3.1: Model parameters and mixing probabilities

Clusters	**Parameters** (a, b)	**Mixing Probabilities**
*	149.35, 20585.11	0.00203055885101554
**	284.30, 394.90	0.00660449752529227
1	165.65, 210.04	0.0353444561749399
2	66.04, 183.90	0.0836724202165458
3	111.35, 183.85	0.0486831288290784
4	34.85, 180.51	0.202374396346127
5	14.89, 150.84	0.374904339750736
6	5.12, 117.34	0.246386202306265

In Figure 3.1 and Table 3.1, we do not assign any cluster number for the first two models but instead we use $*$ and $**$ to refer to these two models, as these two models could be regarded as abnormal models. There are two reasons for this. First, by utilizing data from

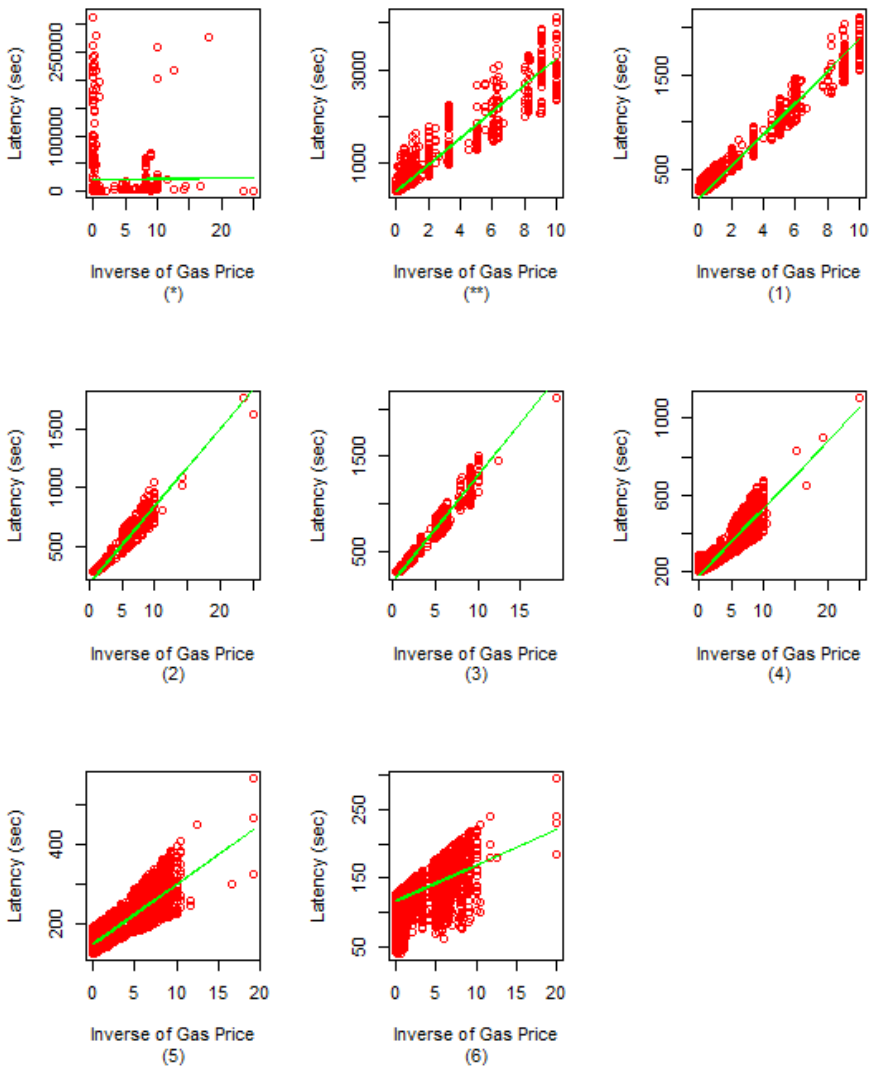


Figure 3.1: Regression models for clusters using finite mixture of linear regressions. Models are listed in descending order of the intercept b.

Etherscan [40], the expectation of block time in Ethereum Mainnet is 15.46 sec based on law of large numbers [41], and the parameter b in the first two models is much higher than that of 12×15.46 as shown in Table 3.1. Second, if we take a look at mixing probabilities of these two models, sum of the mixing probabilities of these two models are less than 0.009, which is way lower than the mixing probability of any other model. Therefore, we could safely take these two models as abnormal models. A potential cause for these two abnormal models might be that transactions from these two models might have exceptional nonce value. For a transaction from these two models, if any other transaction from the same transaction issuer with a lower nonce value is not mined into blockchain, the transaction from these two models have to wait for a long time before mined into blockchain even when its gas price is high.

After excluding the first two models, there are 6 clusters left. According to (3.8), we take the third model in Table 3.1 which has the largest parameter b as the one from the cluster 1. Also, the parameter a for the cluster 1 is 165.65, which is reasonable since it is neither big nor small. For example, for transactions with gas prices as $0.1, 1, 10, 100$ GWei, the corresponding transaction confirmation latency will be $1866.54, 375.69, 226.61, 211.7$ sec, which makes sense based on common knowledge about transaction confirmation latency on Ethereum Mainnet. The reasonableness of the estimates of parameters (a, b) proves the effectiveness of the gas-price vs. transaction-confirmation-latency model.

Since we are using the model from the cluster 1 to approximate our gas price vs. transaction confirmation latency model, the final gas-price vs. transaction-confirmation-latency model is as shown in Figure 3.2.

3.2.3 Simulation Results

Now we have the value of parameters (a, b) in (3.4), we could see how the overall cost changes for different users for a given transaction.

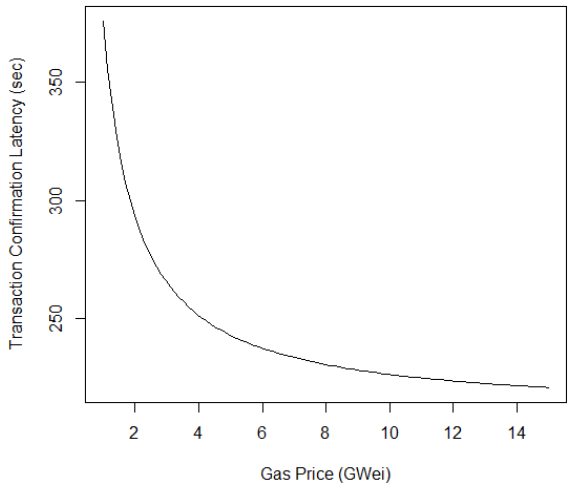


Figure 3.2: Gas-price vs. transaction-confirmation-latency model

Figure 3.3 shows how the overall cost changes with gas price for a given transaction, when users' maximum gas fee and maximum transaction confirmation latency change. Here we take Ethereum token transfer transaction whose gas cost is 21000 as an example. According to the samples from cluster 1 in Figure 3.1, the median gas fee is 0.0282 Ether and the median transaction confirmation latency is 710 sec. Therefore, in the experiment, we choose the following settings: when maximum gas fee is 0.0282 Ether, the maximum transaction confirmation latency is set as 710, 1420, 2130 sec, respectively; when maximum transaction confirmation latency is set as 710 sec, the maximum gas fee is set as 0.0282, 0.0564, 0.0846 Ether, respectively. As shown in Figure 3.3, when maximum gas fee and maximum transaction confirmation latency are fixed, the overall cost will decrease with the increase of gas price before reaching the optimal gas price, and then the overall cost will increase with the increase of gas price after passing the optimal gas price. Particularly, when maximum transaction confirmation latency is fixed, the optimal gas price increases with the increase of maximum gas fee. While, when maximum gas fee is fixed, the optimal gas price decreases with the increase of maximum transaction confirmation latency.

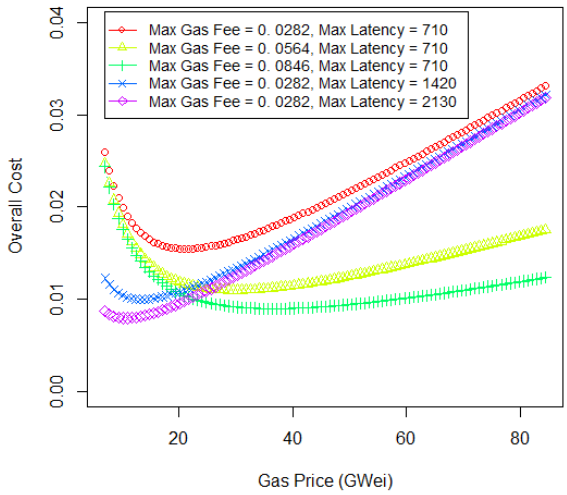


Figure 3.3: Overall cost versus gas price when users change the maximum gas fee and transaction confirmation latency for a given transaction. Here we take gas cost = 21000, $\alpha = 0.5$ as an example.

Figure 3.4 shows how the value of cost function changes with gas price for a given transaction when users' preferences between gas fee and transaction confirmation latency change. For a given α, the overall cost will decrease with the increase of gas price before reaching the optimal gas price, and then the overall cost will increase with the increase of gas price after passing the optimal gas price. When α increases, the optimal gas price to minimize the overall cost decreases. The corresponding optimal gas price is 63.28056, 32.22088, 21.09352, 13.80895, 7.031173 GWei for $\alpha = 0.1, 0.3, 0.5, 0.7, 0.9$, respectively. The reason why the increase of α leads to the decrease of the optimal gas price is that: the bigger α is, the less users want to spend on gas fee, and thus, the less optimal gas price is. Particularly, among all users, the minimal overall cost is the maximum for users with $\alpha = 0.5$, and the same minimal overall cost is shared for two users where the summation of the two users' α equals to 1.

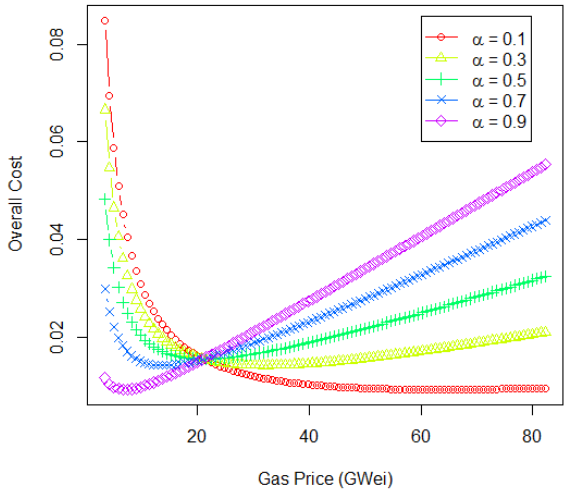


Figure 3.4: Overall cost versus gas price when users' preference between gas fee and transaction confirmation latency changes for a given transaction. Here we take gas cost = 21000, maximum gas fee = 0.0282 Ether, and maximum transaction confirmation latency = 710 sec as an example.

3.3 Experiment

In this section, for evaluating the proposed payment channel based HEX framework, we carry out some experiments by simulating different scenarios in practical token exchange.

3.3.1 Experimental Setup

Experimental Configuration

Hardware and software used in the experiments are listed in Table 3.2 and Table 3.3, respectively.

Table 3.2: List of hardware settings

Device	OS	Memory	CPU
DELL Inspiron 15	Ubuntu 18.04	8G	Intel i5-7300

Table 3.3: List of software settings

Name	Web3.js	Truffle	Ganache	Solidity
Version	1.0.0-beta.35	4.1.13	1.2.2	0.4.24

Baseline Schemes and Performance Metrics

For performance comparison, the used baseline scheme is a conventional HEX similar to DEx.top [22], which does not consider the payment channel. Besides, to evaluate the schemes, the following performance metrics would be used:

- *Gas fee saving ratio*: to describe the percentage of gas fee could be saved by replacing the conventional scheme with the proposed scheme, when the overall cost for a user is minimized in both schemes. Gas fee saving ratio could be expressed as

$$1 - \frac{g_p}{g_c} \tag{3.12}$$

  where

  - g_p denotes the gas fee in the proposed payment channel based HEX when the overall cost for a user is minimized;
  - g_c denotes the gas fee in the conventional HEX when the overall cost for a user is minimized.

- *Latency saving ratio*: to describe the percentage of transaction confirmation latency could be saved by replacing the conventional scheme with the proposed scheme, when the overall cost for a user is minimized in both schemes. Latency saving ratio could be expressed as

$$1 - \frac{l_p}{l_c} \tag{3.13}$$

  where

- l_p denotes the transaction confirmation latency in the proposed payment channel based HEX when the overall cost for a user is minimized;
- l_c denotes the transaction confirmation latency in the conventional HEX when the overall cost for a user is minimized.

- *Blockchain transaction saving ratio*: to describe the percentage of the number of blockchain transactions could be saved by replacing the conventional payment channel with the proposed scheme. Blockchain transaction saving ratio could be expressed as

$$1 - \frac{n_p}{n_c} \tag{3.14}$$

where

- n_p denotes the total number of blockchain transactions in the proposed payment channel based HEX;
- l_c denotes the total number of blockchain transactions in the conventional HEX.

Particularly, to approximate the gas costs of blockchain transactions in both schemes, we implement two MVPs for both schemes. The related code for the MVP of the conventional scheme could be found in Appendix B. The MVP for the proposed scheme has been introduced in Section 2.4. Based on the MVPs, we have simulated the function calls to smart contracts via web3.js to estimate the gas costs of blockchain transactions. Since the gas cost of a blockchain transaction might fluctuate in a small range, we trigger 30 function calls to each target smart contract method and take the average gas cost in 30 simulations as the estimate of the gas cost. Here are the estimated gas costs of transactions to be used in our experiments:

1. In the conventional scheme, the gas cost in a trading transaction is 84841.

2. In the proposed scheme, the gas costs in payment channel creation and closure increase linearly with the increase of the number of types of locked tokens in a payment channel. Table 3.4 shows the related gas costs:

Table 3.4: Gas costs in payment channel creation and closure

Number of Token Types	**Gas Cost in Payment Channel Creation**	**Gas Cost in Payment Channel Closure**
2	266047	73936
5	387892	112216
10	591329	231830
15	794704	359374
20	998020	486756

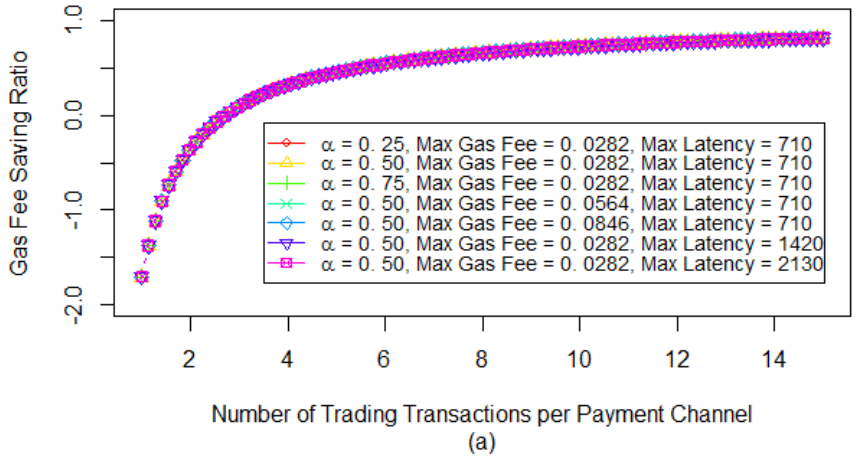
3.3.2 Experimental Cases

As proposed before, the proposed scheme could help save gas fee and transaction confirmation latency for frequent traders, and at the same time ease potential transaction congestion in Ethereum caused by large amounts of trading transactions. Therefore, we consider three types of use cases:

1. How gas fee saving ratio changes with the number of trading transactions per payment channel in terms of different preferences over gas fee and transaction confirmation latency, different limitations for gas fee and transaction confirmation latency, and different gas costs in payment channel creation and closure for different users.

2. How latency saving ratio changes with the number of trading transactions per payment channel in terms of different preferences over gas fee and transaction confirmation latency, different limitations for gas fee and transaction confirmation latency, and different gas costs in payment channel creation and closure for different users.

3. How blockchain transaction saving ratio changes when the average number of trading



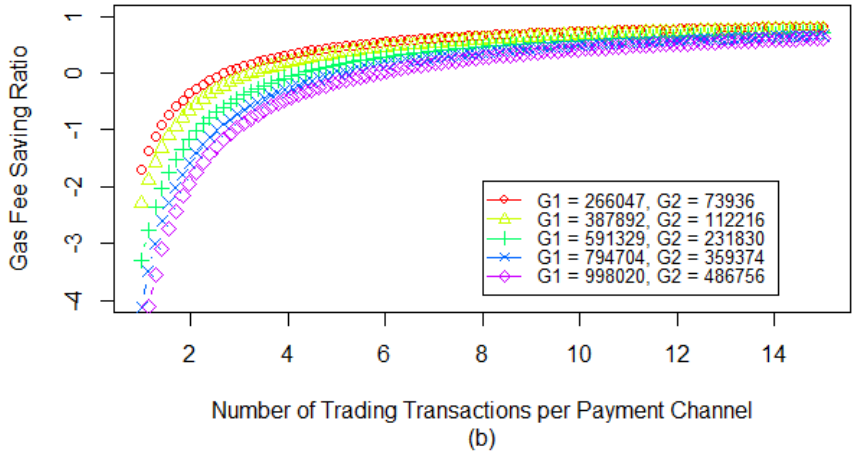


Figure 3.5: Gas fee saving ratio versus number of trading transactions per payment channel.

transactions per payment channel changes for the proposed payment channel based HEX.

How gas fee saving ratio changes

Figure 3.5 (a) shows how gas fee saving ratio changes with the number of trading transactions per payment channel, when users' preferences over gas fee and transaction confirmation latency, maximum gas fee, and maximum transaction confirmation latency change.

Here we take gas costs in payment channel creation and closure as 266047 and 73936 as an example. As shown in Figure 3.5 (a), the gas fee saving ratio increases when the number of tradings per payment channel increases. Particularly, for users whose number of tradings per payment channel is small, they will spend more gas fee in the proposed scheme than in the conventional scheme. This is caused by the overhead of gas fees in creating and closing a payment channel in the proposed scheme. But when users do more tradings per payment channel, the gas fee saving ratio will be larger than 0. Besides, for users with the same amount of trading transactions per payment channel, the gas fee saving ratio does not change when they have different preferences over gas fee and transaction confirmation latency, different maximum gas fee, or different maximum transaction confirmation latency. This proves the fairness of our overall cost function with regards to gas fee saving ratio, which helps different users to obtain the same gas fee saving ratio as long as they have the same number of trading transactions per payment channel.

Figure 3.5 (b) shows how gas fee saving ratio changes with the number of trading transactions per payment channel, when gas costs in payment channel creation and closure increase. In the figure, G1 stands for gas cost in payment channel creation and G2 stands for gas cost in payment channel closure. Here we take $\alpha = 0.5$, maximum gas fee = 0.0282 Ether, and maximum transaction confirmation latency = 710 sec as an example. As shown in Figure 3.5 (b), the gas fee saving ratio will increase when the number of tradings per payment channel increases. Particularly, for users whose number of tradings per payment channel is small, they will spend more gas fee in the proposed scheme than in the conventional scheme. This is caused by the overhead of gas fees in creating and closing a payment channel in the proposed scheme. But when users do more tradings per payment channel, the gas fee saving ratio will be larger than 0. Besides, the higher the summation of the gas costs in payment channel creation and closure is, the higher the

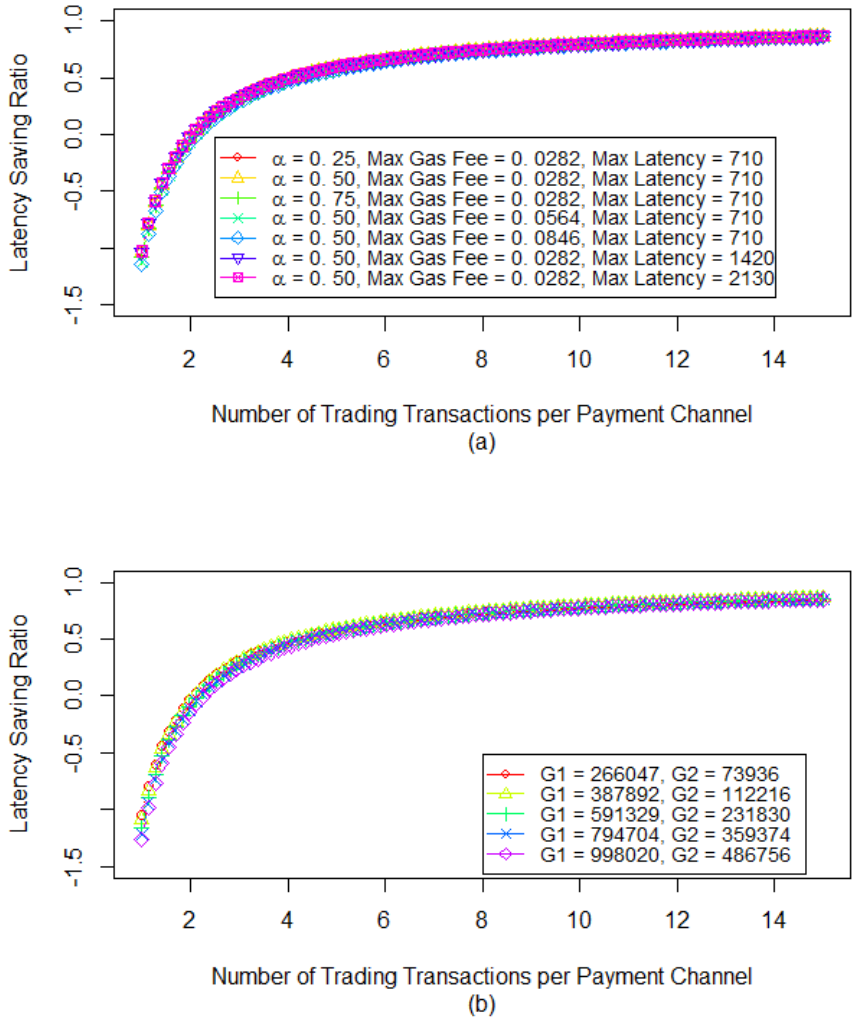


Figure 3.6: Latency saving ratio versus number of trading transactions per payment channel.

number of trading transactions per payment channel is to compensate for the overhead of gas fees in payment channel creation and closure. Particularly, for users with the same amount of trading transactions in the payment channel, the higher the summation of the gas costs in payment channel creation and closure is, the lower the gas fee saving ratio is.

How latency saving ratio changes

Figure 3.6 (a) shows how latency saving ratio changes with the number of trading transactions per payment channel, when users' preferences over gas fee and transaction confirmation latency, maximum gas fee, and maximum transaction confirmation latency change. Here we take gas costs in payment channel creation and closure as 266047 and 73936 as an example. As shown in Figure 3.6 (a), the latency saving ratio will increase with the increase of number of tradings per payment channel. Particularly, for users whose number of tradings per payment channel is less than 2, they will expect longer transaction confirmation latency in the proposed scheme than in the conventional scheme. This is because there are at least two blockchain transactions in the proposed scheme due to payment channel creation and closure. However, as long as users trade more than twice in the payment channel, the latency saving ratio will be larger than 0. Besides, for users with the same amount of tradings in the payment channel, there is only a minor difference in the latency saving ratio when they have different preferences over gas fee and transaction confirmation latency, different maximum gas fee, or different maximum transaction confirmation latency. This proves the fairness of our overall cost function with regards to latency saving ratio, which helps different users to obtain similar latency saving ratio as long as they have the same number of trading transactions per payment channel.

Figure 3.6 (b) shows how latency saving ratio changes with the number of trading transactions per payment channel, when gas costs in payment channel creation and closure increase. In the figure, G1 stands for gas cost in payment channel creation and G2 stands for gas cost in payment channel closure. Here we take $\alpha = 0.5$, maximum gas fee = 0.0282 Ether, and maximum transaction confirmation latency = 710 sec as an example. As shown in Figure 3.6 (b), the latency saving ratio will increase when the number of tradings per payment channel increases. Particularly, for users whose number of tradings

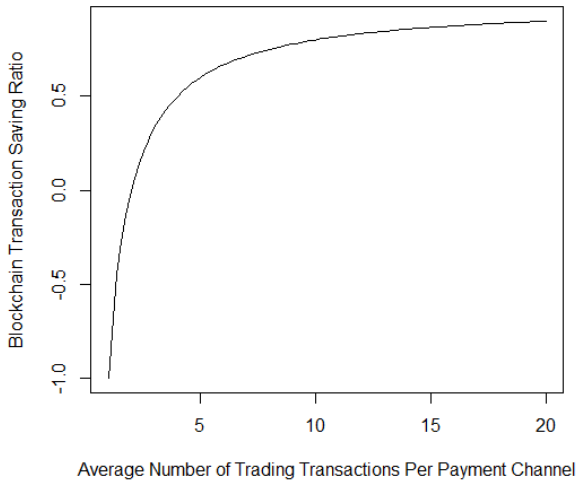


Figure 3.7: Blockchain transaction saving ratio versus average number of trading transactions per payment channel.

per payment channel is less than 2, they will expect longer transaction confirmation latency in the proposed scheme than in the conventional scheme. This is because there are at least two blockchain transactions in the proposed scheme due to payment channel creation and closure. However, as long as users trade more than twice in a payment channel, the latency saving ratio will be larger than 0. Besides, for users with the same amount of tradings in the payment channel, the higher the summation of gas costs in payment channel creation and closure is, the lower the latency saving ratio is, though the difference is quite minor.

How blockchain transaction saving ratio changes

As shown in Figure 3.7, the blockchain transaction saving ratio increases when the average number of tradings per payment channel for all users increases in the proposed scheme. Particularly, when the average number of tradings per payment channel for the proposed scheme is less than 2, the blockchain transaction saving ratio will be less than 0. This is because there are at least two blockchain transactions in the proposed scheme due to payment channel creation and closure. However, as long as users trade more than twice in

average, the more often users choose the proposed scheme over the conventional scheme, the more on-chain transactions could be saved, which helps ease potential transaction congestion in Ethereum caused by trading transactions in HEX.

3.4 Summary

In this chapter, we quantitatively evaluated the performance of payment channel in HEX by comparisons of the cost in terms of gas fee and transaction confirmation latency for a user in the conventional HEX and the proposed payment channel based HEX when the user's overall cost is minimized.

First, we showed how the overall cost could be minimized. We proposed the very first gas-price vs. transaction-confirmation-latency function. Based on the gas-price vs. transaction-confirmation-latency function, we used the weighted sum of gas fee and transaction confirmation latency to model the overall cost for a HEX user and formulated the overall cost minimization problem as a convex optimization problem. Due to the unknown parameters in the proposed gas-price vs. transaction-confirmation-latency function, we collected 540,296 blockchain transactions from the Ethereum Mainnet and introduced finite mixture of linear regressions to get qualitative estimates of these unknown parameters based on the collected transactions. The reasonableness of the estimates of these unknown parameters demonstrated the effectiveness of the proposed gas-price vs. transaction-confirmation-latency function. Our problem formulation of the overall cost minimization took into account the user's preference between gas fee and transaction confirmation latency, the maximum gas fee, and the maximum transaction confirmation latency. Simulation results showed how the overall cost changes with gas price when the HEX user's preference between gas fee and transaction confirmation latency, the maximum gas fee, and the maximum transaction confirmation latency change.

Second, based on the overall cost function, we compared the cost in terms of gas fee and transaction confirmation latency for a user in the conventional HEX and the proposed HEX when the user's overall cost is minimized. Also we compared the total number of blockchain transactions needed for all users in the conventional HEX and the proposed HEX. Experimental results demonstrated the effectiveness of our proposed payment channel based HEX in terms of reducing the total gas fee and transaction confirmation latency for frequent traders as well as the potential transaction congestion in Ethereum.

Chapter 4

Conclusions and Future Work

In this thesis, we firstly propose a systematic design of the proposed payment channel based HEX which benefits frequent Ethereum token traders and alleviates the potential transaction congestion caused by excessive trading transactions in Ethereum. Afterward, we propose the very first gas-price vs. transaction-confirmation-latency model to help blockchain transaction issuers to minimize the overall cost of gas fee and transaction confirmation latency. Based on the proposed gas-price vs. transaction-confirmation-latency function, we quantitatively evaluate the performance of the proposed payment channel based HEX. Experimental results have been presented to show that our proposed solution incurs a low overhead while achieving a high efficiency.

While the proposed platform is promising, this thesis reveals several limitations that we intend to address in our future work to refine this platform:

1. Due to the nature of token lock in creating payment channel, malicious HEX users may initiate attacks to HEX by creating a large number of channels, or creating a large number of cancelled orders. Therefore, an effective incentive-and-punishment mechanism or KYC might be needed to prevent these attacks.

2. Implementation of the proposed system needs to be optimized. As shown from our experimental results, gas fees in payment channel creation and closure increase lin-

early with the number of locked tokens' types. An optimized implementation is required to lower down gas fees on payment channel creation and closure when the number of locked tokens' types is huge.

3. More data may be helpful to get more accurate qualitative estimates of parameters in the gas-price vs. transaction-confirmation-latency model. Due to high cost in terms of money in setting up Ethereum full nodes in AWS, we are only able to collect Ethereum transactions for three days for now. If we could afford to collecting more data in the future, we might be able to compare the results from data sets in different dates and get more accurate parameter estimates in the gas-price vs. transaction-confirmation-latency model.

Bibliography

[1] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung, "Decentralized applications: The blockchain-empowered software system," *IEEE Access*, vol. 6, no. 99, pp. 53 019–53 033, Oct. 2018.

[2] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, Vienna, Austria, Oct. 2016.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf [Accessed: 06/04/2019].

[4] G. WOOD, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, Apr. 2014.

[5] U. Mukhopadhyay, A. Skjellum, O. Hambolu, J. Oakley, L. Yu, and R. Brooks, "A brief survey of cryptocurrency systems," in *Int. Conf. Privacy, Security and Trust (PST)*, Auckland, New Zealand, Dec. 2016.

[6] CoinMarketCap, "Top 100 cryptocurrencies by market capitalization." [Online]. Available: https://coinmarketcap.com [Accessed: 06/01/2018].

[7] N. Álvarez Díaz, J. Herrera-Joancomartí, and P. Caballero-Gil, "Smart contracts based on blockchain for logistics management," in *Proc. 1st Int. Conf. Internet of Things and Maching Learning*, Liverpool, United Kingdom, Oct. 2017.

[8] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," Jan. 2016. [Online]. Available: https://lightning.network/lightning-network-paper.pdf [Accessed: 06/04/2019].

[9] B. Xiao, X. Fan, S. Gao, and W. Cai, "Edgetoll: A blockchain-based toll collection system for public sharing of heterogeneous edges," in *Proc. IEEE Int. Conf. Computer Communications Workshops*, Paris, France, Apr. 2019.

[10] A. Feder, N. Gandal, J. T. Hamrick, and T. Moore, "The impact of DDoS and other security shocks on Bitcoin currency exchanges: evidence from Mt. Gox," *Journal of Cybersecurity*, vol. 3, no. 2, pp. 137–144, 01 2018.

[11] "Coinbase," Jan. 2018. [Online]. Available: https://www.coinbase.com/ [Accessed: 06/04/2019].

[12] "Gemini," Feb. 2018. [Online]. Available: https://gemini.com/ [Accessed: 06/04/2019].

[13] "Poloniex," Feb. 2018. [Online]. Available: https://poloniex.com/ [Accessed: 06/04/2019].

[14] "Kraken," April. 2018. [Online]. Available: https://www.kraken.com/ [Accessed: 06/04/2019].

[15] "Huobi," March. 2018. [Online]. Available: https://www.huobi.co [Accessed: 06/04/2019].

[16] L. Luu and Y. Velner, "Kybernetwork: A trustless decentralized exchange and payment service," *White Paper*, Aug. 2017. [Online]. Available: https://whitepaper.io/document/43/kyber-network-whitepaper [Accessed: 06/04/2019].

[17] M. Oved and D. Mosites, "Swap: A peer-to-peer protocol for trading ethereum tokens," *White Paper*, Jun. 2017. [Online]. Available: https://whitepaperdatabase.com/airswap-ast-whitepaper/ [Accessed: 06/04/2019].

[18] "Etherdelta," May. 2018. [Online]. Available: https://github.com/etherdelta [Accessed: 06/04/2019].

[19] W. Warren and A. Bandeali, "0x: An open protocol for decentralized exchange on the ethereum blockchain," *White Paper*, Feb. 2017. [Online]. Available: https://0xproject.com/pdfs/0x_white_paper.pdf [Accessed: 06/04/2019].

[20] Aurora Labs, "Idex: A real-time and high-throughput ethereum smart contract exchange," *White Paper*, Nov. 2017. [Online]. Available: https://idex.market/static/IDEX-Whitepaper-V0.7.5.pdf [Accessed: 06/04/2019].

[21] JOYSO, "Joyso: World's first decentralized exchange," *White Paper*, Nov. 2018. [Online]. Available: https://joyso.io/wp-content/uploads/2018/11/JOYSO-whitepaper.pdf [Accessed: 06/04/2019].

[22] DEx.top, "Dex technical white paper," Jun. 2018. [Online]. Available: https://github.com/dexDev/DEx.top/blob/master/whitepaper/DEx-Whitepaper-Short-Version.pdf [Accessed: 08/04/2018].

[23] P. Todd, "Op_checklocktimeverify," Oct. 2014. [Online]. Available: https://https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki [Accessed: 06/04/2019].

[24] C. Decker and R. Wattenhofer, "A fast and scalable payment network with bitcoin duplex micropayment channels," in *Proc. 17th Int. Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Edmonton, Canada, Aug. 2015.

[25] G. Avarikioti, F. Laufenberg, J. Sliwinski, Y. Wang, and R. Wattenhofer, "Towards secure and efficient payment channels," *CoRR*, vol. abs/1811.12740, 2018.

[26] S. Eskandari, S. Moosavi, and J. Clark, "Sok: Transparent dishonesty: front-running attacks on blockchain," *arXiv preprint arXiv:1902.05164*, 2019.

[27] "Know your customer," Sep. 2018. [Online]. Available: https://en.wikipedia.org/wiki/Know_your_customer [Accessed: 06/04/2019].

[28] Ethereum, "Solidity documentation release 0.4.24." [Online]. Available: https://solidity.readthedocs.io/en/v0.4.24/ [Accessed: 06/01/2018].

[29] T. team, "Truffle overview." [Online]. Available: https://www.trufflesuite.com/docs/truffle/overview [Accessed: 06/01/2018].

[30] F. Vogelsteller, M. Kotewicz, J. Wilcke, and M. Oancea, "web3.js documentation release 1.0.0," Jul. 2019. [Online]. Available: https://buildmedia.readthedocs.org/media/pdf/web3js/v1.2.0/web3js.pdf [Accessed: 08/04/2019].

[31] Hwang, Ching-Lai, S. M. Masud, and Abu, *Multiple Objective Decision Making—Methods and Applications: A State-of-the-Art Survey*, 1st ed. Springer-Verlag Berlin Heidelberg, Jan. 1979, vol. 164.

[32] P. C. Fishburn, "Additive utilities with incomplete product sets: Application to priorities and assignments," *Operations Research*, vol. 15, pp. 537–542, 1967.

[33] O. Grodzevich and O. Romanko, "Normalization and other topics in multi-objective optimization," in *Proc. the Fields-MITACS Industrial Problems Workshop*, Toronto, Canada, Aug. 2006.

[34] F. P. Miller, A. F. Vandome, and J. McBrewster, *Amazon Web Services*. Alpha Press, 2010.

[35] R. D. D. VEAUX, "Mixtures of linear regressions," *Comput. Stat. Data Anal.*, vol. 8, no. 3, pp. 227–245, Nov. 1989.

[36] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the royal statistical society, series B*, vol. 39, no. 1, pp. 1–38, 1977.

[37] C. Fraley and A. E. Raftery, "Model-based clustering, discriminant analysis, and density estimation," *Journal of the American Statistical Association*, vol. 97, no. 458, pp. 611–631, 2002.

[38] T. Benaglia, D. Chauveau, D. R. Hunter, and D. Young, "mixtools: An R package for analyzing finite mixture models," *Journal of Statistical Software*, vol. 32, no. 6, pp. 1–29, 2009.

[39] D. Karlis and E. Xekalaki, "Choosing initial values for the em algorithm for finite mixtures," *Comput. Stat. Data Anal.*, vol. 41, no. 3-4, pp. 577–590, Jan. 2003.

[40] Etherscan, "Ethereum block time history." [Online]. Available: https://etherscan.io/chart/blocktime [Accessed: 06/04/2019].

[41] G. Grimmett and D. Stirzaker, *Probability and Random Processes*, 3rd ed. Oxford university press, 2003, vol. 80.

Appendices

Appendix A - Smart contract for the payment channel based HEX

```
1  pragma solidity ^0.4.24;
2  import "./Token.sol";
3  import "./Token1.sol";
4
5  contract HEXWithPC is SafeMath {
6      // the admin address
7      address public admin;
8
9      // starts with 0, which means ETH
10     address[] supportedTokens;
11
12     uint256 numChannels;
13
14     //mapping of token addresses to mapping of account balances (token=0
           means Ether)
15     mapping (address => mapping (address => uint)) public tokens;
16
17     //record available Tokens of HEX, mapping of token addresses to
           mapping of account balances (token=0 means Ether)
18     mapping (address => mapping (address => uint)) public availableTokens
           ;
19
20      //mapping of channel id to lockedTokens
```

```
21      mapping (bytes32 => mapping (address => mapping (address => uint)))
            public
22      lockedTokens;
23
24      // mapping of token addrss to its scale factor compared with ETH
25      mapping (address => uint8) public scaleFactors;
26
27      // mapping of users to their related payment channel ids;
28      mapping (address => bytes32) public channelUsers;
29
30      // maping of Payment Channel id to Payment Channel
31      mapping (bytes32 => Channel) channels;
32
33      enum ChannelStatus{
34        Open,
35        Challenged,
36        Closed
37      }
38
39      struct Channel{
40        address sender;
41        address recipent;
42        uint256 challengePeriod;
43        uint256 challengePeriodLength;
44        uint256 nounce;
45        uint256 expire;
46        ChannelStatus status;
47      }
48
49       event Deposit(address token, address user, uint amount, uint balance
            );
```

```
50      event Withdraw(address token, address user, uint amount, uint
            balance);
51
52      event CreatePC(bytes32 pc_id, address sender);
53      event ClosePC(bytes32 pc_id, address sender);
54
55      event Verify(address signer, address sender);
56
57      constructor (address admin_){
58          admin = admin_;
59      }
60
61      modifier onlyAdmin() {
62          require(msg.sender == admin);
63          _;
64      }
65
66      function getAdmin() returns (address){
67          return admin;
68      }
69
70      // get the id of the payment channel
71      function getPCId() returns (bytes32){
72          return channelUsers[msg.sender];
73      }
74
75      function getPCStatus(bytes32 id) returns (ChannelStatus) {
76          return channels[id].status;
77      }
78
79      function deposit() payable {
```

```
80          tokens[0][msg.sender] = safeAdd(tokens[0][msg.sender], msg.value)
                ;
81          availableTokens[0][msg.sender] = safeAdd(availableTokens[0][msg.
                sender],msg.value);
82          emit Deposit(0, msg.sender, msg.value, tokens[0][msg.sender]);
83      }
84
85      function depositToken(address token, uint amount) {
86          // remember to call Token(address).approve(this, amount) or this
                contract will not be able to do the transfer on your behalf.
87          if (token==0) revert();
88          if (!Token(token).transferFrom(msg.sender, this, amount)) revert
                ();
89          tokens[token][msg.sender] = safeAdd(tokens[token][msg.sender],
                amount);
90          availableTokens[token][msg.sender] = safeAdd(availableTokens[
                token][msg.sender],amount);
91          emit Deposit(token, msg.sender, amount, tokens[token][msg.sender
                ]);
92        }
93
94      function withdraw(uint amount) {
95          // check if any payment channel is open
96          require(channelUsers[msg.sender] == 0x0, 'User has a live channel
                !');
97
98          // check if enough amount of tokens for withdraw
99          if (availableTokens[0][msg.sender] < amount) revert();
100
101         tokens[0][msg.sender] = safeSub(tokens[0][msg.sender], amount);
```

```
102         availableTokens[0][msg.sender] = safeSub(availableTokens[0][msg.
                sender], amount);
103
104         if (!msg.sender.call.value(amount)()) revert();
105         emit Withdraw(0, msg.sender, amount, tokens[0][msg.sender]);
106       }
107
108     function withdrawToken(address token, uint amount) {
109         if (token==0) revert();
110
111         // check if any payment channel is open
112         require(channelUsers[msg.sender] == 0x0, 'User has a live channel
                !');
113
114         if (availableTokens[token][msg.sender] < amount) revert();
115
116         tokens[token][msg.sender] = safeSub(tokens[token][msg.sender],
                amount);
117         availableTokens[token][msg.sender] = safeSub(availableTokens[
                token][msg.sender], amount);
118
119         if (!Token(token).transfer(msg.sender, amount)) revert();
120
121         emit Withdraw(token, msg.sender, amount, tokens[token][msg.sender
                ]);
122       }
123
124     // create a payment channel via a two-party signed message
125     // @param: sender represents hex user
126     // @param: token represents the deposited token address by the user
```

```
127     // @param: amount represents the number of referred token deposited
            by the user
128
129     function createPC(address maker, uint256[] makerBalance, uint256[]
            takerBalance, address[] tokenAddr, uint8[] v, bytes32[] r,
            bytes32[] s) returns (bytes32){
130       // only one channel per user
131     //require(taker == admin, 'Taker has to be admin');
132       require(channelUsers[maker] == 0x0, 'User already has a live
              channel!');
133         require(makerBalance.length == tokenAddr.length, 'Invalid user
                balance');
134         require(takerBalance.length == tokenAddr.length, 'Invalid admin
                balance');
135
136
137         bytes32 hash = sha3(maker, makerBalance, admin, takerBalance,
                tokenAddr);
138         // verify user signature
139         require(verify(hash, maker, v[0], r[0],s[0]), "Message was not
                signed by user");
140         // verify admin signature
141         require(verify(hash, admin, v[1], r[1],s[1]), "Message was not
                signed by admin");
142
143       // create an id for the new channel
144       numChannels++;
145       bytes32 _id = keccak256(now + numChannels);
146
147         // address targetToken;
148         uint8 i = 0;
```

```
149
150         for(i = 0; i < makerBalance.length; i++){
151             address targetToken = tokenAddr[i];
152             require(makerBalance[i] >= 0 && takerBalance[i] >= 0, "
                    invalid makerBalance or takerBalance");
153
154             lockedTokens[_id][targetToken][maker] = safeAdd(makerBalance[
                    i], 0);
155             availableTokens[targetToken][maker] = safeSub(availableTokens
                    [targetToken][maker], makerBalance[i]);
156
157       lockedTokens[_id][targetToken][admin] = safeAdd(takerBalance[i], 0)
           ;
158         availableTokens[targetToken][admin] = safeSub(availableTokens[
              targetToken][admin],takerBalance[i]);
159
160         }
161
162       // map the newly created channel to the user and to the id of the
           channel
163
164         Channel memory _channel = Channel(
165         maker,
166         admin,
167         0,
168         3,
169         0,
170         30,
171         ChannelStatus.Open
172         );
173
```

```
174         channels[_id] = _channel;
175           channelUsers[maker] = _id;
176
177         emit CreatePC(_id,maker);
178         return _id;
179       }
180
181
182       // closePC when both parties agree
183       // @param: maker represents traders
184       // @param: taker represents the HEx's admin account
185       // @param: balance records the token ammouts distributed to the maker
              and the taker, ordered by the creation time of tokens
186       function closePC(bytes32 id, address maker, address taker, uint256[]
              makerBalance,
187             uint256[] takerBalance, address[] tokenAddr, uint8[] v, bytes32
                  [] r, bytes32[] s) public {
188         // check if related payment channel for the user exists or not
189         require(channelUsers[maker] == id,  "Channel ID does not belongs to
               the user");
190       require(taker == admin, "taker has to be the admin");
191
192         // check if related input for balance is valid
193         require(makerBalance.length == takerBalance.length,"Array length
              conflicts!");
194         require(makerBalance.length == tokenAddr.length, 'Invalid user
              balance');
195         require(takerBalance.length == tokenAddr.length, 'Invalid admin
              balance');
196
```

```
197         bytes32 hash = sha3(id, maker, makerBalance, taker, takerBalance,
                tokenAddr);
198
199         // verify user signature
200         require(verify(hash, maker, v[0], r[0],s[0]), "Message was not
                signed by user");
201
202         // verify admin signature
203         require(verify(hash, taker, v[1], r[1],s[1]), "Message was not
                signed by admin");
204
205         // release locked funds
206         for(uint8 i = 0; i < makerBalance.length ; i++){
207               address targetToken = tokenAddr[i];
208
209               // make sure the sum of the tokens from both parties is
                      consistent during the live time of the payment channel
210               require(makerBalance[i] >= 0 && takerBalance[i] >= 0, "
                      invalid makerBalance or takerBalance");
211               require(makerBalance[i] + takerBalance[i] == lockedTokens[id
                      ][targetToken][maker] + lockedTokens[id][targetToken][
                      taker], "input balance does not match with locked token
                      balance");
212
213              // reset the token amounts of the trader
214               availableTokens[targetToken][maker] = makerBalance[i];
215               tokens[targetToken][maker] = makerBalance[i];
216
217               // reset the token amounts of the admin
218               availableTokens[targetToken][taker] = safeAdd(availableTokens
                      [targetToken][taker],takerBalance[i]);
```

```
219             tokens[targetToken][taker] = availableTokens[targetToken][
                    taker];
220       }
221
222       // remove the mapping of between user and the payment channel
223       delete channels[id];
224       delete channelUsers[maker];
225       numChannels--;
226
227       emit ClosePC(id,maker);
228     }
229
230     function addSupportedTokens (address token, uint8 scaleFactor)
            onlyAdmin{
231       supportedTokens.push(token);
232       scaleFactors[token] = scaleFactor;
233      }
234
235      function getAvailableToken (address token) returns(uint256){
236        return availableTokens[token][msg.sender];
237      }
238
239      function getLockedToken (address token) returns(uint256){
240        return tokens[token][msg.sender] - availableTokens[token][msg.
               sender];
241      }
242
243      function getLockedTokenByPCId (bytes32 id, address token) returns(
           uint256){
244        return lockedTokens[id][token][msg.sender];
245      }
```

```
246
247     function getSuppportedTokenNumber () returns(uint256){
248       return supportedTokens.length;
249     }
250
251     function getTokenAmount (address token) returns (uint){
252         return tokens[token][msg.sender];
253     }
254
255     // return the balance of all supported tokens for the msg.sender
256     function getBalance() returns (uint[]){
257       uint[] balance;
258       address token;
259
260       for(uint8 i = 0; i < supportedTokens.length; i++){
261         token = supportedTokens[i];
262         balance.push(tokens[token][msg.sender]);
263       }
264
265       return balance;
266     }
267        /* @dev check if the provided signature is valid, internal
268        * @param hash signed information
269        * @param sender signer address
270        * @param v sig_v
271        * @param r sig_r
272        * @param s sig_s
273        */
274        function verify(bytes32 hash, address sender, uint8 v, bytes32 r,
               bytes32 s) returns (bool) {
```

```
275             return ecrecover(keccak256("\x19Ethereum Signed Message:\n32",
                    hash), v, r, s) == sender;
276         }
277
278         function GetSha32(bytes32 id, address maker, uint[] makerBalance,
                 address taker, uint[] takerBalance, address[] tokensAddr )
                returns (bytes32) {
279             return sha3(id, maker, makerBalance, taker, takerBalance,
                    tokensAddr);
280         }
281
282         function GetSha31(address maker, uint256[] makerBalance, address
                taker, uint256[] takerBalance, address[] tokensAddr) returns
                (bytes32) {
283             return sha3(maker, makerBalance, taker, takerBalance,
                    tokensAddr);
284         }
285         /* function GetSha31(address maker, uint256[] makerBalance,
                address taker, uint256[] takerBalance) returns (bytes32) {
286             return sha3(maker, makerBalance, taker, takerBalance);
287         }*/
288 }
```

Appendix B - Smart contract for the conventional HEX

```
1  pragma solidity 0.4.24;
2  import "./Token.sol";
3
4  contract HEX is SafeMath {
5      // the admin address
6       address public admin;
7
8      // starts with 0, which means ETH
9       address[] supportedTokens;
10
11     // mapping of token addresses to mapping of account balances (token=0
           means Ether)
12      mapping (address => mapping (address => uint)) public tokens;
13
14      event Deposit(address token, address user, uint amount, uint balance
           );
15      event DepositToken(address token, address user, uint amount, uint
           balance);
16
17      event Withdraw(address token, address user, uint amount, uint
           balance);
18      event SettleOrder(address makerToken, address maker, uint
           makerBalance, address takerToken, address taker, uint
           takerBalance);
19
20
21     constructor (address admin_){
22         admin = admin_;
23     }
```

```
24
25      modifier onlyAdmin() {
26          require(msg.sender == admin);
27          _;
28      }
29
30      function getAdmin() returns (address){
31          return admin;
32      }
33
34      function deposit() payable {
35          tokens[0][msg.sender] = safeAdd(tokens[0][msg.sender], msg.value)
                ;
36
37          emit Deposit(0, msg.sender, msg.value, tokens[0][msg.sender]);
38        }
39
40      function depositToken(address token, uint amount) {
41          // remember to call Token(address).approve(this, amount) or this
                contract will not be able to do the transfer on your behalf.
42          if (token==0) revert();
43          if (!Token(token).transferFrom(msg.sender, this, amount)) revert
                ();
44          tokens[token][msg.sender] = safeAdd(tokens[token][msg.sender],
                amount);
45
46          emit DepositToken(token, msg.sender, amount, tokens[token][msg.
                sender]);
47        }
48
49
```

```
50      function withdraw(uint amount) {
51          if (tokens[0][msg.sender] < amount) revert();
52          tokens[0][msg.sender] = safeSub(tokens[0][msg.sender], amount);
53          if (!msg.sender.call.value(amount)()) revert();
54
55          emit Withdraw(0, msg.sender, amount, tokens[0][msg.sender]);
56        }
57
58      function withdrawToken(address token, uint amount) {
59          if (token==0) revert();
60          if (tokens[token][msg.sender] < amount) revert();
61          tokens[token][msg.sender] = safeSub(tokens[token][msg.sender],
                amount);
62          if (!Token(token).transfer(msg.sender, amount)) revert();
63
64          emit Withdraw(token, msg.sender, amount, tokens[token][msg.sender
                ]);
65        }
66
67    // Deprecated, use settleOrder2 instead
68       function settleOrder(address maker, address makerToken, uint
              makerBalance, address takerToken, uint takerBalance,
69           address taker, address takerToken2, uint takerBalance2, address
                 makerToken2, uint makerBalance2) onlyAdmin returns (bool) {
70          // compare the selling order and buying order, make sure both
                orders match each other
71          require(makerToken == makerToken2);
72          require(takerToken == takerToken2);
73          require(makerBalance == makerBalance2);
74          require(takerBalance == takerBalance2);
75
```

```
76          // substract the amount of tokens maker and taker used for the
                order
77          tokens[makerToken][maker] = safeSub(tokens[makerToken][maker],
                makerBalance);
78          tokens[takerToken][taker] = safeSub(tokens[takerToken][taker],
                takerBalance);
79
80          // add the amount of tokens maker and taker gained for the order
81          tokens[takerToken][maker] = safeAdd(tokens[takerToken][maker],
                takerBalance);
82          tokens[makerToken][taker] = safeAdd(tokens[makerToken][taker],
                makerBalance);
83
84          emit SettleOrder(takerToken,maker,takerBalance,makerToken,taker,
                makerBalance);
85
86          return true;
87      }
88
89
90      function settleOrder2 (address makerToken, uint makerBalance,
            address takerToken, uint takerBalance,
91            address[] user, uint8[] v, bytes32[] r, bytes32[] s) onlyAdmin
                returns (bool) {
92
93
94          // check the validity of parameters
95          require(user.length == 3, "invalid traders");
96          require(admin == user[2], "order not signed by admin");
97
98          // check the signature from the maker order
```

```
99          // bytes32 hash = sha256(makerToken, makerBalance, takerToken,
                takerBalance);
100         bytes32 hash = sha3(makerToken, makerBalance, takerToken,
                takerBalance);
101         require(verify(hash, user[0], v[0], r[0], s[0]), "Message not
                signed by maker");
102
103         // check the signature from the selling order
104         require(verify(hash, user[1], v[1], r[1], s[1]), "Message not
                signed by taker");
105
106         // check the signature from the selling admin
107         require(verify(hash, user[2], v[2], r[2], s[2]), "Message not
                signed by admin");
108
109         address maker = user[0];
110         address taker = user[1];
111
112         // substract the amount of tokens maker and taker used for the
                order
113         tokens[makerToken][maker] = safeSub(tokens[makerToken][maker],
                makerBalance);
114         tokens[takerToken][taker] = safeSub(tokens[takerToken][taker],
                takerBalance);
115
116         // add the amount of tokens maker and taker gained for the order
117         tokens[takerToken][maker] = safeAdd(tokens[takerToken][maker],
                takerBalance);
118         tokens[makerToken][taker] = safeAdd(tokens[makerToken][taker],
                makerBalance);
119
```

```
120         emit SettleOrder(makerToken, maker, makerBalance, takerToken,
                taker, takerBalance);
121
122         return true;
123     }
124
125     function getTokenAmount (address token) returns (uint){
126         return tokens[token][msg.sender];
127     }
128
129     function addSupportedTokens (address token) onlyAdmin{
130       supportedTokens.push(token);
131     }
132
133     function getSuppportedTokens () returns(uint256){
134       return supportedTokens.length;
135     }
136
137    /* @dev check if the provided signature is valid, internal
138    * @param hash signed information
139    * @param sender signer address
140    * @param v sig_v
141    * @param r sig_r
142    * @param s sig_s
143    */
144    function verify(bytes32 hash, address sender, uint8 v, bytes32 r,
          bytes32 s) pure returns (bool) {
145       return ecrecover(keccak256("\x19Ethereum Signed Message:\n32",
              hash), v, r, s) == sender;
146    }
147 }
```